ldcpy

Release 0.17.post2+dirty

Alex Pinard, Allison Baker, Anderson Banihirwe, Dorit Hammerlin

Oct 25, 2022

CONTENTS:

ldcpy is a utility for gathering and plotting metrics from NetCDF or Zarr files using the Pangeo stack. It also contains a number of statistical and visual tools for gathering metrics and comparing Earth System Model data files.

AUTHORS
Alex Pinard, Allison Baker, Anderson Banihirwe, Dorit Hammerling

LICENSE
Apache 2.0

Documentation and usage examples are available here.

CHAPTER

ONE

REFERENCE TO LDCPY PAPER

A. Pinard, D. M. Hammerling, and A. H. Baker. Assessing differences in large spatiotemporal climate datasets with a new Python package. In The 2020 IEEE International Workshop on Big Data Reduction, 2020. doi: 10.1109/BigData50022.2020.9378100.

Link to paper: https://doi.org/10.1109/BigData50022.2020.9378100

1.1 Installation using Conda (recommended)

Ensure conda is up to date and create a clean Python (3.6+) environment:

```
conda update conda
conda create --name ldcpy python=3.8
conda activate ldcpy
```

Now install ldcpy:

```
conda install -c conda-forge ldcpy
```

1.2 Alternative Installation

Ensure pip is up to date, and your version of python is at least 3.6:

```
pip install --upgrade pip
python --version
```

Install cartopy using the instructions provided at https://scitools.org.uk/cartopy/docs/latest/installing.html.

Then install ldcpy:

```
pip install ldcpy
```

1.3 Accessing the tutorial

If you want access to the tutorial notebook, clone the repository (this will create a local repository in the current directory):

```
git clone https://github.com/NCAR/ldcpy.git
```

Start by enabling Hinterland for code completion and code hinting in Jupyter Notebook and then opening the tutorial notebook:

```
jupyter nbextension enable hinterland/hinterland
jupyter notebook
```

The tutorial notebook can be found in docs/source/notebooks/TutorialNotebook.ipynb, feel free to gather your own metrics or create your own plots in this notebook!

Other example notebooks that use the sample data in this repository include PopData.ipynb and MetricsNotebook.ipynb.

The AWSDataNotebook grabs data from AWS, so can be run on a laptop with the caveat that the files are large.

The following notebooks asume that you are using NCAR's JupyterHub (https://jupyterhub.hpc.ucar.edu): LargeDataGladenotebook.ipynb, CAMNotebook,ipynb, and error_bias.ipynb

1.4 Re-create notebooks with Pangeo Binder

Try the notebooks hosted in this repo on Pangeo Binder. Note that the session is ephemeral. Your home directory will not persist, so remember to download your notebooks if you make changes that you need to use at a later time!

Note: All example notebooks are in docs/source/notebooks (the easiest ones to use in binder first are TutorialNotebook.ipynb and PopData.ipynb)

CHAPTER

TWO

DOCUMENTATION

2.1 Installation

2.1.1 Installation using Conda (recommended)

Ensure conda is up to date and create a clean Python (3.6+) environment:

```
conda update conda
conda create --name ldcpy python=3.8
conda activate ldcpy
```

Now install ldcpy:

```
conda install -c conda-forge ldcpy
```

2.1.2 Alternative Installation

Ensure pip is up to date, and your version of python is at least 3.6:

```
pip install --upgrade pip
python --version
```

Install cartopy using the instructions provided at https://scitools.org.uk/cartopy/docs/latest/installing.html.

Then install ldcpy:

```
pip install ldcpy
```

2.1.3 Accessing the tutorial (for users)

If you want access to the tutorial notebook, clone the repository (this will create a local repository in the current directory):

```
git clone https://github.com/NCAR/ldcpy.git
```

Start by activating the ldcpy environment, enabling Hinterland for code completion in Jupyter Notebook and then starting the notebook server:

```
conda activate ldcpy
conda install -c conda-forge jupyter_contrib_nbextensions
jupyter contrib nbextension install --user
jupyter nbextension enable hinterland/hinterland
jupyter notebook
```

The tutorial notebook can be found in docs/source/notebooks/TutorialNotebook.ipynb, feel free to gather your own metrics or create your own plots in this notebook!

2.2 Development

2.2.1 Installation for Developers

First, clone the repository and cd into the root of the repository:

```
git clone https://github.com/NCAR/ldcpy.git
cd ldcpy
```

For a development install, do the following in the ldcpy repository directory:

```
conda env update -f environment.yml
conda activate ldcpy
python -m pip install -e .
```

Then install the pre-commit script and git hooks for code style checking:

```
pre-commit install
```

This code block enables optional extensions for code completion, code hinting and minimizing tracebacks in Jupyter. Then start the jupyter notebook server in your browser (at localhost:8888):

```
jupyter nbextension enable hinterland/hinterland
jupyter nbextension enable skip-traceback/main

conda activate ldcpy
jupyter notebook
```

2.2.2 Instructions and Tips for Contributing

For viewing changes to documentation in the repo, do the following:

```
pip install -r docs/requirements.txt
sphinx-reload docs/
```

This starts and opens a local version of the documentation in your browser (at localhost:5500/index.html) and keeps it up to date with any changes made. Note that changes to docstrings in the code will not trigger an update, only changes to the .rst files in the docs/ folder.

If you have added a feature or fixed a bug, add new tests to the appropriate file in the tests/ directory to test that the feature is working or that the bug is fixed. Before committing changes to the code, run all tests from the project root directory to ensure they are passing.

```
pytest -n 4
```

Additionally, rerun the TutorialNotebook in Jupyter (Kernel -> Restart & Run All). Check that no unexpected behavior is encountered in these plots.

Now you are ready to commit your code. pre-commit should automatically run black, flake8, and isort to enforce style guidelines. If changes are made, the first commit will fail and you will need to stage the changes that have been made before committing again. If, for some reason, pre-commit fails to make changes to your files, you should be able to run the following to clean the files manually:

```
black --skip-string-normalization --line-length=100 .
flake8 .
isort .
```

2.2.3 Adding new package dependencies to ldcpy

1) Adding new package dependencies requires updating the code in the following four places:

    /ci/environment.yml /ci/environment-dev.yml /ci/upstream-dev-environment.yml /requirements.txt

If the package dependency is specifically used for documentation, instead of adding it to /requirements.txt, add it to:

    /docs/source/requirements.txt

If this package is only used for documentation, skip the remaining steps.

2) If the package is one that includes C code (such as numpy or scipy), update the autodoc_mock_imports list in /docs/source/conf.py. The latest build of the documentation can be found at (https://readthedocs.org/projects/ldcpy/builds/), if the build fails and the error message indicates a problem with the newest package - try adding it to autodoc_mock_imports.

3) Finally, update the ldcpy-feedstock repository (git clone https://github.com/conda-forge/ldcpy-feedstock.git), or manually create a branch and add the dependency in the browser. Name the branch add-<new_dependency_name>. In the file /recipe/meta.yaml, in the "requirements" section, under "run", add your dependency to the list.

4) If the CI build encounters errors after adding a dependency, check the status of the CI workflow at (https://github.com/NCAR/ldcpy/actions?query=workflow%3ACI) to determine if the error is related to the new package.

2.2.4 Creating a Release

Updating the package on PyPi:

1) On the ldcpy Github page, select Releases on the right sidebar, and select "Draft a new release"

2) Create a new tag by incrementing the minor or major version number. Give the release a title and description.

3) Publish the release. Check the Actions tab -> Upload Packageg to PyPi workflow to ensure it completes.

Updating the package on Conda Forge:

0) Ensure the package has been updated on PyPi.

1) Fork the ldcpy_feedstock repository (https://github.com/conda-forge/ldcpy-feedstock)

2) In recipe/meta.yml, set the version number to match the latest release tag. Make sure the build number is 0 if you are changing the version number.

3) In recipe/meta.yml, update the sha256 hash. The hash for the latest release can be found at https://pypi.org/project/ldcpy/#files. Copy the hash from ldcpy-x.xx.xx.tar.gz.

4) In recipe/meta.yml, add any new package dependencies under the run section of the requirements.
5) From your fork's github page, create a pull request pointed at the conda_forge repository.
6) Make sure each step listed in the pull request checklist is completed. See https://conda-forge.org/docs/maintainer/updating_pkgs.html if needed.
7) Allow some time for all the tests to complete, these take between 8-20 minutes. See the error/warning output if any tests fail.
8) Merge the pull request. The new version will be available on conda forge shortly.

2.3 API Reference

This page provides an auto-generated summary of ldcpy's API. For more details and examples, refer to the relevant chapters in the main part of the documentation.

2.3.1 ldcpy Util (ldcpy.util)

ldcpy.util.**check_metrics**(*ds*, *varname*, *set1*, *set2*, *ks_tol=0.05*, *pcc_tol=0.99999*, *spre_tol=5.0*, *ssim_tol=0.995*, ***calcs_kwargs*)

Check the K-S, Pearson Correlation, and Spatial Relative Error calcs

Parameters

- **ds** (*xarray.Dataset*) – An xarray dataset containing multiple netCDF files concatenated across a 'collection' dimension
- **varname** (*str*) – The variable of interest in the dataset
- **set1** (*str*) – The collection label of the "control" data
- **set2** (*str*) – The collection label of the (1st) data to compare
- **ks_tol** (*float, optional*) – The p-value threshold (significance level) for the K-S test (default = .05)
- **pcc_tol** (*float, optional*) – The default Pearson corrolation coefficient (default = .99999)
- **spre_tol** (*float, optional*) – The percentage threshold for failing grid points in the spatial relative error test (default = 5.0).
- **ssim_tol** (*float, optional*) – The threshold for the data ssim test (default = .995
- ****calcs_kwargs** – Additional keyword arguments passed through to the Datasetcalcs instance.

Returns
out

Return type
Number of failing calcs

Notes

Check the K-S, Pearson Correlation, and Spatial Relative Error calcs from:

A. H. Baker, H. Xu, D. M. Hammerling, S. Li, and J. Clyne, "Toward a Multi-method Approach: Lossy Data Compression for Climate Simulation Data", in J.M. Kunkel et al. (Eds.): ISC High Performance Workshops 2017, Lecture Notes in Computer Science 10524, pp. 30–42, 2017 (doi:10.1007/978-3-319-67630-2_3).

Check the Data SSIM, which is a modification of SSIM calc from:

A.H. Baker, D.M. Hammerling, and T.L. Turton. "Evaluating image quality measures to assess the impact of lossy data compression applied to climate simulation data", Computer Graphics Forum 38(3), June 2019, pp. 517-528 (doi:10.1111/cgf.13707).

Default tolerances for the tests are:

K-S: fail if p-value < .05 (significance level) Pearson correlation coefficient: fail if coefficient < .99999 Spatial relative error: fail if > 5% of grid points fail relative error Data SSIM: fail if Data SSIM < .995

ldcpy.util.**collect_datasets**(*data_type*, *varnames*, *list_of_ds*, *labels*, ***kwargs*)

Concatonate several different xarray datasets across a new "collection" dimension, which can be accessed with the specified labels. Stores them in an xarray dataset which can be passed to the ldcpy plot functions (Call this OR open_datasets() before plotting.)

Parameters

- **data_type** (*string*) – Current data types: :cam-fv, pop
- **varnames** (*list*) – The variable(s) of interest to combine across input files (usually just one)
- **list_of_datasets** (*list*) – The datasets to be concatonated into a collection
- **labels** (*list*) – The respective label to access data from each dataset (also used in plotting fcns)

  **kwargs : (optional) – Additional arguments passed on to xarray.concat(). A list of available arguments can be found here: https://xarray-test.readthedocs.io/en/latest/generated/xarray.concat.html

Returns

out – a collection containing all the data from the list datasets

Return type

xarray.Dataset

ldcpy.util.**compare_stats**(*ds*, *varname: str*, *sets*, *significant_digits: int = 5*, *include_ssim: bool = False*, *weighted: bool = True*, ***calcs_kwargs*)

Print error summary statistics for multiple DataArrays (should just be a single time slice)

Parameters

- **ds** (*xarray.Dataset*) – An xarray dataset containing multiple netCDF files concatenated across a 'collection' dimension
- **varname** (*str*) – The variable of interest in the dataset
- **sets** (*list of str*) – The labels of the collection to compare (all will be compared to the first set)
- **significant_digits** (*int, optional*) – The number of significant digits to use when printing stats (default 5)

- **include_ssim** (*bool, optional*) – Whether or not to compute the image ssim - slow for 3D vars (default: False)
- **weighted** (*bool, optional*) – Whether or not weight the means (default = True)
- ****calcs_kwargs** – Additional keyword arguments passed through to the Datasetcalcs instance.

Returns
out

Return type
None

ldcpy.util.**open_datasets**(*data_type*, *varnames*, *list_of_files*, *labels*, *weights=True*, ***kwargs*)

Open several different netCDF files, concatenate across a new 'collection' dimension, which can be accessed with the specified labels. Stores them in an xarray dataset which can be passed to the ldcpy plot functions.

Parameters

- **data_type** (*string*) – Current data types: :cam-fv, pop
- **varnames** (*list*) – The variable(s) of interest to combine across input files (usually just one)
- **list_of_files** (*list*) – The file paths for the netCDF file(s) to be opened
- **labels** (*list*) – The respective label to access data from each netCDF file (also used in plotting fcns)
- ****kwargs** – (optional) – Additional arguments passed on to xarray.open_mfdataset(). A list of available arguments can be found here: http://xarray.pydata.org/en/stable/generated/xarray.open_dataset.html

Returns
out – a collection containing all the data from the list of files

Return type
xarray.Dataset

ldcpy.util.**save_metrics**(*full_ds*, *varname*, *set1*, *set2*, *time=0*, *lev=0*, *location='names.csv'*)

full_ds
[xarray.Dataset] An xarray dataset containing multiple netCDF files concatenated across a 'collection' dimension

varname
[str] The variable of interest in the dataset

set1
[str] The collection label of the "control" data

set2
[str] The collection label of the (1st) data to compare

time
[int, optional] The time index used t (default = 0)

time
[lev, optional] The level index of interest in a 3D dataset (default 0)

Returns
out

Return type
Number of failing metrics

ldcpy.util.**subset_data**(*ds*, *subset=None*, *lat=None*, *lon=None*, *lev=None*, *start=None*, *end=None*, *time_dim_name='time'*, *vertical_dim_name=None*, *lat_coord_name=None*, *lon_coord_name=None*)

Get a subset of the given dataArray, returns a dataArray

2.3.2 ldcpy Plot (ldcpy.plot)

class ldcpy.plot.**calcsPlot**(*ds*, *varname*, *calc*, *sets*, *group_by=None*, *scale='linear'*, *calc_type='raw'*, *plot_type='spatial'*, *transform='none'*, *subset=None*, *approx_lat=None*, *approx_lon=None*, *lev=0*, *color='coolwarm'*, *standardized_err=False*, *quantile=None*, *contour_levs=24*, *short_title=False*, *axes_symmetric=False*, *legend_loc='upper right'*, *vert_plot=False*, *tex_format=False*, *legend_offset=None*, *weighted=True*, *basic_plot=False*, *cmax=None*, *cmin=None*)

This class contains code to plot calcs in an xarray Dataset that has either 'lat' and 'lon' dimensions, or a 'time' dimension.

time_series_plot(*da_sets*, *titles*)

time series plot

ldcpy.plot.**plot**(*ds*, *varname*, *calc*, *sets*, *group_by=None*, *scale='linear'*, *calc_type='raw'*, *plot_type='spatial'*, *transform='none'*, *subset=None*, *lat=None*, *lon=None*, *lev=0*, *color='coolwarm'*, *quantile=None*, *start=None*, *end=None*, *short_title=False*, *axes_symmetric=False*, *legend_loc='upper right'*, *vert_plot=False*, *tex_format=False*, *legend_offset=None*, *weighted=True*, *basic_plot=False*, *cmax=None*, *cmin=None*)

Plots the data given an xarray dataset

Parameters

- **ds** (*xarray.Dataset*) – The input dataset
- **varname** (*str*) – The name of the variable to be plotted
- **calc** (*str*) – The name of the calc to be plotted (must match a property name in the Datasetcalcs class in ldcpy.plot, for more information about the available calcs see ldcpy.Datasetcalcs) Acceptable values include:
  - ns_con_var
  - ew_con_var
  - mean
  - std
  - variance
  - prob_positive
  - prob_negative
  - odds_positive
  - zscore
  - mean_abs

  - mean_squared
  - rms
  - sum
  - sum_squared
  - corr_lag1
  - quantile
  - lag1
  - standardized_mean
  - ann_harmonic_ratio
  - pooled_variance_ratio
- **sets** (*list <str>*) – The labels of the dataset to gather calcs from
- **group_by** (*str*) – how to group the data in time series plots. Valid groupings:
  - time.day
  - time.dayofyear
  - time.month
  - time.year
- **scale** (*str, optional*) – time-series y-axis plot transformation. (default "linear") Valid options:
  - linear
  - log
- **calc_type** (*str, optional*) – The type of operation to be performed on the calcs. (default 'raw') Valid options:
  - raw: the unaltered calc values
  - diff: the difference between the calc values in the first set and every other set
  - ratio: the ratio of the calc values in (2nd, 3rd, 4th... sets/1st set)
  - calc_of_diff: the calc value computed on the difference between the first set and every other set
- **plot_type** (*str , optional*) – The type of plot to be created. (default 'spatial') Valid options:
  - spatial: a plot of the world with values at each lat and lon point (takes the mean across the time dimension)
  - time-series: A time-series plot of the data (computed by taking the mean across the lat and lon dimensions)
  - histogram: A histogram of the time-series data
- **transform** (*str, optional*) – data transformation. (default 'none') Valid options:
  - none
  - log

- **subset** (*str, optional*) – subset of the data to gather calcs on (default None). Valid options:
  - first5: the first 5 days of data
  - DJF: data from the months December, January, February
  - MAM: data from the months March, April, May
  - JJA: data from the months June, July, August
  - SON: data from the months September, October, November
- **lat** (*float, optional*) – The latitude of the data to gather calcs on (default None).
- **lon** (*float , optional*) – The longitude of the data to gather calcs on (default None).
- **lev** (*float, optional*) – The level of the data to gather calcs on (used if plotting from a 3d data set), (default 0).
- **color** (*str, optional*) – The color scheme for spatial plots, (default 'coolwarm'). see https://matplotlib.org/3.1.1/gallery/color/colormap_reference.html for more options
- **quantile** (*float, optional*) – A value between 0 and 1 required if calc="quantile", corresponding to the desired quantile to gather, (default 0.5).
- **start** (*int, optional*) – A value between 0 and the number of time slices indicating the start time of a subset, (default None).
- **end** (*int, optional*) – A value between 0 and the number of time slices indicating the end time of a subset, (default None)
- **calc_ssim** (*bool, optional*) – Whether or not to calculate the ssim (structural similarity index) between two plots (only applies to plot_type = 'spatial'), (default False)
- **short_title** (*bool, optional*) – If True, use a shortened title in the plot output (default False).
- **axes_symmetric** (*bool, optional*) – Whether or not to make the colorbar axes symmetric about zero (used in a spatial plot) (default False)
- **legend_loc** (*str, optional*) – The location to put the legend in a time-series plot in single-column format (plot_type = "time_series", vert_plot=True) (default "upper right")
- **vert_plot** (*bool, optional*) – If true, forces plots into a single column format and enlarges text. (default False)
- **tex_format** (*bool, optional*) – Whether to interpret all plot output strings as latex formatting (default False)
- **legend_offset** (*2-tuple, optional*) – The x- and y- offset of the legend. Moves the corner of the legend specified by legend_loc to the specified location specified (where (0,0) is the bottom left corner of the plot and (1,1) is the top right corner). Only affects time-series, histogram, and periodogram plots.

Returns
out

Return type
None

ldcpy.plot.**tex_escape**(*text*)

Parameters
text – a plain text message

Returns
the message escaped to appear correctly in LaTeX

2.3.3 ldcpy Metrics (ldcpy.metrics)

class ldcpy.calcs.**Datasetcalcs**(*ds: DataArray*, *data_type: str*, *aggregate_dims: list*, *time_dim_name: str = 'time'*, *lat_dim_name: Optional[str] = None*, *lon_dim_name: Optional[str] = None*, *vert_dim_name: Optional[str] = None*, *lat_coord_name: Optional[str] = None*, *lon_coord_name: Optional[str] = None*, *q: float = 0.5*, *weighted=True*)

This class contains calcs for each point of a dataset after aggregating across one or more dimensions, and a method to access these calcs. Expects a DataArray.

get_calc(*name: str*, *q: Optional[int] = 0.5*, *grouping: Optional[str] = None*, *ddof=1*)

Gets a calc aggregated across one or more dimensions of the dataset

Parameters

- **name** (*str*) – The name of the calc (must be identical to a property name)
- **q** (*float, optional*) – (default 0.5)

Returns
out – A DataArray of the same size and dimensions the original dataarray, minus those dimensions that were aggregated across.

Return type
xarray.DataArray

get_single_calc(*name: str*)

Gets a calc consisting of a single float value

Parameters
name (*str*) – the name of the calc (must be identical to a property name)

Returns
out – The calc value

Return type
float

property annual_harmonic_relative_ratio: DataArray

The annual harmonic relative to the average periodogram value in a neighborhood of 50 frequencies around the annual frequency NOTE: This assumes the values along the "time" dimension are equally spaced. NOTE: This calc returns a lat-lon array regardless of aggregate dimensions, so can only be used in a spatial plot.

property annual_harmonic_relative_ratio_pct_sig: ndarray

The percentage of points past the significance cutoff (p value <= 0.01) for the annual harmonic relative to the average periodogram value in a neighborhood of 50 frequencies around the annual frequency

property cdf: DataArray

The empirical CDF of the dataset.

property entropy: DataArray

An estimate for the entropy of the data (using gzip) # lower is better (1.0 means random - no compression possible)

property ew_con_var: DataArray

The East-West Contrast Variance averaged along the aggregate dimensions

property lag1: DataArray

The deseasonalized lag-1 autocorrelation value by day of year NOTE: This calc returns an array of spatial values as the data set regardless of aggregate dimensions, so can only be plotted in a spatial plot.

property lag1_first_difference: DataArray

The deseasonalized lag-1 autocorrelation value of the first difference of the data by day of year NOTE: This calc returns an array of spatial values as the data set regardless of aggregate dimensions, so can only be plotted in a spatial plot.

property lat_autocorr: DataArray

the correlation of a variable with itself shifted in the latitude dimension

Type
Autocorrelation

property lev_autocorr: DataArray

the correlation of a variable with itself shifted in the vertical dimension

Type
Autocorrelation

property lon_autocorr: DataArray

the correlation of a variable with itself shifted in the longitude dimension

Type
Autocorrelation

property mae_day_max: DataArray

The day of maximum mean absolute value at the point. NOTE: only available in spatial and spatial comparison plots

property mean: DataArray

The mean along the aggregate dimensions

property mean_abs: DataArray

The mean of the absolute errors along the aggregate dimensions

property mean_squared: DataArray

The absolute value of the mean along the aggregate dimensions

property most_repeated: DataArray

Most repeated value in dataset

property most_repeated_percent: DataArray

Most repeated value in dataset

property n_s_first_differences: DataArray

First differences along the west-east direction

property ns_con_var: DataArray

The North-South Contrast Variance averaged along the aggregate dimensions

property num_negative: DataArray

The probability that a point is negative

property num_positive: DataArray

The probability that a point is positive

property num_zero: DataArray

The probability that a point is zero

property odds_positive: DataArray

The odds that a point is positive = prob_positive/(1-prob_positive)

property percent_unique: DataArray

Percentage of unique values in the dataset

property pooled_variance: DataArray

The overall variance of the dataset

property pooled_variance_ratio: DataArray

The pooled variance along the aggregate dimensions

property prob_negative: DataArray

The probability that a point is negative

property prob_positive: DataArray

The probability that a point is positive

property range: DataArray

The range of the dataset

property root_mean_squared: DataArray

The absolute value of the mean along the aggregate dimensions

property standardized_mean: DataArray

The mean at each point along the aggregate dimensions divided by the standard deviation NOTE: will always be 0 if aggregating over all dimensions

property std: DataArray

The standard deviation along the aggregate dimensions

property variance: DataArray

The variance along the aggregate dimensions

property w_e_derivative: DataArray

Derivative of dataset from west-east

property w_e_first_differences: DataArray

First differences along the west-east direction

property zscore: DataArray

The z-score of a point averaged along the aggregate dimensions under the null hypothesis that the true mean is zero. NOTE: currently assumes we are aggregating along the time dimension so is only suitable for a spatial plot.

property zscore_cutoff: ndarray

The Z-Score cutoff for a point to be considered significant

property zscore_percent_significant: ndarray

The percent of points where the zscore is considered significant

class ldcpy.calcs.**Diffcalcs**(*ds1: DataArray*, *ds2: DataArray*, *data_type: str*, *aggregate_dims: Optional[list] = None*, *spre_tol: float = 0.0001*, *k1: float = 0.01*, *k2: float = 0.03*, ***calcs_kwargs*)

This class contains calcs on the overall dataset that require more than one input dataset to compute

get_diff_calc(*name: str*, *color: Optional[str] = 'coolwarm'*)

Gets a calc on the dataset that requires more than one input dataset

Parameters
name (*str*) – The name of the calc (must be identical to a property name)

Returns
out

Return type
float

property covariance: DataArray

The covariance between the two datasets

property ks_p_value

The Kolmogorov-Smirnov p-value

property max_spatial_rel_error

We compute the relative error at each grid point and return the maximun.

property normalized_max_pointwise_error

The absolute value of the maximum pointwise difference, normalized by the range of values for the first set

property normalized_root_mean_squared

The absolute value of the mean along the aggregate dimensions, normalized by the range of values for the first set

property pearson_correlation_coefficient

returns the pearson correlation coefficient between the two datasets

property spatial_rel_error

At each grid point, we compute the relative error. Then we report the percentage of grid point whose relative error is above the specified tolerance (1e-4 by default).

property ssim_value

We compute the SSIM (structural similarity index) on the visualization of the spatial data. This creates two plots and uses the standard SSIM.

property ssim_value_fp_fast

Faster implementation then ssim_value_fp_slow (this is the default DSSIM option).

property ssim_value_fp_slow

We compute the SSIM (structural similarity index) on the spatial data - using the data itself (we do not create an image) - this is the slower non-matrix implementation that is good for experiementing (not in practice).

2.4 Examples

2.4.1 Tutorial

ldcpy is a utility for gathering and plotting derived quantities from NetCDF or Zarr files using the Pangeo stack. This tutorial notebook targets comparing CESM data in its original form to CESM data that has undergone lossy compression (meaning that the reconstructed file is not exactly equivalent to the original file). The tools provided in ldcpy are intended to highlight differences due to compression artifacts in order to assist scientist in evaluating the amount of lossy compression to apply to their data.

The CESM data used here are NetCDF files in "timeseries" file format, meaning that each NetCDF file contains one (major) output variable (e.g., surface temperature or precipitation rate) and spans multiple timesteps (daily, monthly, 6-hourly, etc.). CESM timeseries files are regularly released in large public datasets.

[1]:
```
# Add ldcpy root to system path
import sys

sys.path.insert(0, '../../../')

# Import ldcpy package
# Autoreloads package everytime the package is called, so changes to code will be␣
↪reflected in the notebook if the above sys.path.insert(...) line is uncommented.
%load_ext autoreload
%autoreload 2

# suppress all of the divide by zero warnings
import warnings

warnings.filterwarnings("ignore")

import ldcpy

# display the plots in this notebook
%matplotlib inline
```

Overview

This notebook demonstrates the use of ldcpy on the sample data included with this package. It explains how to open datasets (and view metadata), display basic statistics about the data, and create both time-series and spatial plots of the datasets and related quantitiess. Plot examples start out with the essential arguments, and subsequent examples explore the additional plotting options that are available.

For information about installation, see these instructions, and for information about usage, see the API reference here.

Loading Datasets and Viewing Metadata

The first step in comparing the data is to load the data from the files that we are interested into a "collection" for ldcpy to use. To do this, we use ldcpy.open_datasets(). This function requires the following three arguments:

- *varnames* : the variable(s) of interest to combine across files (typically the timeseries file variable name)
- *list_of_files* : a list of full file paths (either relative or absolute)
- *labels* : a corresponding list of names (or labels) for each file in the collection

Note: This function is a wrapper for xarray.open_mfdatasets(), and any additional key/value pairs passed in as a dictionary are used as arguments to xarray.open_mfdatasets(). For example, specifying the chunk size ("chunks") will be important for large data (see LargeDataGladeNotebook.ipynb for more information and an example).

We setup three different collections of timeseries datasets in these examples:

- *col_ts* contains daily surface temperature (TS) data (2D data) for 100 days
- *col_prect* contains daily precipitation rate (PRECT) data (2D data) for 60 days
- *col_t* contains monthly temperature (T) data (3D data) for 3 months

These datasets are collections of variable data from several different netCDF files, which are given labels in the third parameter to the ldcpy.open_datasets() function. These names/labels can be whatever you want (e.g., "orig", "control", "bob", ...), but they should be informative because the names will be used to select the appropriate dataset later and as part of the plot titles.

In this example, in each dataset collection we include a file with the original (uncompressed) data as well as additional file(s) with the same data subject to different levels of lossy compression.

Note: If you do not need to get the data from files (e.g., you have already used xarray.open_dataset()), then use ldcpy.collect_datasets() instead of ldcpy.open_datasets (see example in AWSDataNotebook.ipynb).

[2]:
```
# col_ts is a collection containing TS data
col_ts = ldcpy.open_datasets(
    "cam-fv",
    ["TS"],
    [
        "../../../data/cam-fv/orig.TS.100days.nc",
        "../../../data/cam-fv/zfp1.0.TS.100days.nc",
        "../../../data/cam-fv/zfp1e-1.TS.100days.nc",
    ],
    ["orig", "zfpA1.0", "zfpA1e-1"],
)
# col_prect contains PRECT data
col_prect = ldcpy.open_datasets(
    "cam-fv",
    ["PRECT"],
    [
        "../../../data/cam-fv/orig.PRECT.60days.nc",
        "../../../data/cam-fv/zfp1e-7.PRECT.60days.nc",
        "../../../data/cam-fv/zfp1e-11.PRECT.60days.nc",
    ],
    ["orig", "zfpA1e-7", "zfpA1e-11"],
)
# col_t contains 3D T data (here we specify the chunk to be a single timeslice)
col_t = ldcpy.open_datasets(
    "cam-fv",
```

(continues on next page)

(continued from previous page)

```
    ["T"],
    [
        "../../../data/cam-fv/cam-fv.T.3months.nc",
        "../../../data/cam-fv/c.fpzip.cam-fv.T.3months.nc",
    ],
    ["orig", "comp"],
    chunks={"time": 1},
)
```

```
dataset size in GB 0.07

dataset size in GB 0.04

dataset size in GB 0.04
```

Note that running the open_datasets function (as above) prints out the size of each dataset collection. For *col_prect* , the chunks parameter is used by DASK (which is further explained in LargeDataGladeNotebook.ipynb).

Printing a dataset collection reveals the dimension names, sizes, datatypes and values, among other metadata. The dimensions and the length of each dimension are listed at the top of the output. Coordinates list the dimensions vertically, along with the data type of each dimension and the coordinate values of the dimension (for example, we can see that the 192 latitude data points are spaced evenly between -90 and 90). Data variables lists all the variables available in the dataset. For these timeseries files, only the one (major) variable will be of interest. For *col_t* , that variable is temperature (T), which was specified in the first argument of the open_datasets() call. The so-called major variable will have the required "lat", "lon", and "time" dimensions. If the variable is 3D (as in this example), a "lev" dimension will indicates that the dataset contains values at multiple altitudes (here, lev=30). Finally, a "collection" dimension indicates that we concatenated several datasets together. (In this *col_t* example, we concatonated 2 files together.)

[3]:
```
# print information about col_t
col_t
```

[3]:
```
<xarray.Dataset>
Dimensions:     (collection: 2, time: 3, lev: 30, lat: 192, lon: 288)
Coordinates:
  * lat         (lat) float64 -90.0 -89.06 -88.12 -87.17 ... 88.12 89.06 90.0
  * lev         (lev) float64 3.643 7.595 14.36 24.61 ... 957.5 976.3 992.6
  * lon         (lon) float64 0.0 1.25 2.5 3.75 5.0 ... 355.0 356.2 357.5 358.8
  * time        (time) object 1920-02-01 00:00:00 ... 1920-04-01 00:00:00
    cell_area   (lat, collection, lon) float64 dask.array<chunksize=(192, 1, 288),␣
↪meta=np.ndarray>
  * collection  (collection) <U4 'orig' 'comp'
Data variables:
    T           (collection, time, lev, lat, lon) float32 dask.array<chunksize=(1, 1, 30,
↪ 192, 288), meta=np.ndarray>
Attributes: (12/15)
    Conventions:      CF-1.0
    source:           CAM
    case:             b.e11.B20TRC5CNBDRD.f09_g16.031
    title:            UNSET
    logname:          mickelso
    host:             ys0219
```

(continues on next page)

(continued from previous page)

```
...                 ...
topography_file:    /glade/p/cesmdata/cseg/inputdata/atm/cam/topo/USGS-gtop...
history:            Thu Jul  9 14:15:11 2020: ncks -d time,0,2,1 cam-fv.T.6...
NCO:                netCDF Operators version 4.7.9 (Homepage = http://nco.s...
cell_measures:      area: cell_area
data_type:          cam-fv
file_size:          {'orig': 10622798, 'comp': 2994433}
```

Comparing Summary Statistics

The compare_stats function can be used to compute and compare the overall statistics for a **single** timeslice in two (or more) datasets from a collection. To use this function, three arguments are required. In order, they are:

- *ds* - a single time slice in a collection of datasets read in from ldcpy.open_datasets() or ldcpy.collect_datasets()
- *varname* - the variable name we want to get statistics for (in this case 'TS' is the variable in our dataset collection *col_ts*)
- *sets* - the labels of the datasets in the collection we are interested in cpmparing (if more than two, all are compare to he first one)

Additionally, three optional arguments can be specified:

- *significant_digits* - the number of significant digits to print (default = 5)
- *include_ssim* - include the image ssim (default = False. Note this takes a bit of time for 3D vars)
- *weighted* - use weighted averages (default = True)

[4]:
```
# print 'TS' statistics about 'orig', 'zfpA1.0', and 'zfpA1e-1 and diff between the 'orig' and␣
↪the other two datasets
# for time slice = 0
ds = col_ts.isel(time=0)
ldcpy.compare_stats(ds, "TS", ["orig", "zfpA1.0", "zfpA1e-1"], significant_digits=6)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

	orig	zfpA1.0	zfpA1e-1
mean	284.49	284.481	284.489
variance	534.015	533.69	533.995
standard deviation	23.1088	23.1017	23.1083
min value	216.741	216.816	216.747
min (abs) nonzero value	216.741	216.816	216.747
max value	315.584	315.57	315.576
probability positive	1	1	1
number of zeros	0	0	0
spatial autocorr - latitude	0.993918	0.993911	0.993918
spatial autocorr - longitude	0.996801	0.996791	0.996801
entropy estimate	0.414723	0.247491	0.347534

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

	zfpA1.0	zfpA1e-1
max abs diff	0.405884	0.0229187
min abs diff	0	0
mean abs diff	0.0565128	0.0042215
mean squared diff	7.32045e-05	3.04995e-07
root mean squared diff	0.0729538	0.00532525
normalized root mean squared diff	0.000761543	5.39821e-05
normalized max pointwise error	0.00410636	0.00023187
pearson correlation coefficient	0.999995	1
ks p-value	1	1
spatial relative error(% > 0.0001)	68.958	0
max spatial relative error	0.00147397	8.11948e-05
DSSIM	0.981813	0.998227
file size ratio	1.87	1.28

[5]:
```
# without weighted means
ldcpy.compare_stats(ds, "TS", ["orig", "zfpA1.0", "zfpA1e-1"], significant_digits=6,␣
↪weighted=False)
```

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

	orig	zfpA1.0	zfpA1e-1
mean	274.714	274.708	274.713
variance	534.006	533.681	533.985
standard deviation	23.1088	23.1017	23.1083
min value	216.741	216.816	216.747
min (abs) nonzero value	216.741	216.816	216.747
max value	315.584	315.57	315.576
probability positive	1	1	1
number of zeros	0	0	0
spatial autocorr - latitude	0.993918	0.993911	0.993918
spatial autocorr - longitude	0.996801	0.996791	0.996801
entropy estimate	0.414723	0.247491	0.347534

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

	zfpA1.0	zfpA1e-1
max abs diff	0.405884	0.0229187
min abs diff	0	0
mean abs diff	0.0585202	0.00422603
mean squared diff	3.32617e-05	1.51646e-07
root mean squared diff	0.075273	0.00533574
normalized root mean squared diff	0.000761543	5.39821e-05
normalized max pointwise error	0.00410636	0.00023187
pearson correlation coefficient	0.999995	1
ks p-value	1	1
spatial relative error(% > 0.0001)	68.958	0
max spatial relative error	0.00147397	8.11948e-05
DSSIM	0.981813	0.998227
file size ratio	1.87	1.28

We can also generate derived quantities on a particular dataset. While this is done "behind the scenes" with the plotting functions, we first demonstrate here how the user can access this data without creating a plot.

We use an object of type ldcpy.Datasetcalcs to gather calculations derived from a dataset. To create an ldcpy.DatasetMetircs object, we first grab the particular dataset from our collection that we are interested in (in the usual xarray manner). For example, the following will grab the data for the TS variable labeled 'orig' in the *col_ts* dataset that we created:

[6]:
```
# get the orig dataset
my_data = col_ts["TS"].sel(collection="orig")
my_data.attrs["data_type"] = col_ts.data_type
my_data.attrs["set_name"] = "orig"
```

Then we create a Datasetcalcs object using the data and a list of dimensions that we want to aggregate the data along. We usually want to aggregate all of the timeslices ("time") or spatial points ("lat" and "lon"):

[7]:
```
ds_calcs_across_time = ldcpy.Datasetcalcs(my_data, "cam-fv", ["time"])
ds_calcs_across_space = ldcpy.Datasetcalcs(my_data, "cam-fv", ["lat", "lon"])
```

Now when we call the get_calc() method on this class, a quantity will be computed across each of these specified dimensions. For example, below we compute the "mean" across time.

[8]:
```
my_data_mean_across_time = ds_calcs_across_space.get_calc_ds("mean", "m")

# trigger computation
my_data_mean_across_time
```

[8]:
```
<xarray.Dataset>
Dimensions:     (time: 100)
Coordinates:
  * time        (time) object 1920-01-01 00:00:00 ... 1920-04-10 00:00:00
    collection  <U8 'orig'
Data variables:
    m           (time) float64 dask.array<chunksize=(100,), meta=np.ndarray>
Attributes:
    units:          K
    long_name:      Surface temperature (radiative)
    cell_methods:   time: mean
    data_type:      cam-fv
    set_name:       orig
    cell_measures:  area: cell_area
```

[9]:
```
# Here just ask for the spatial mean at the first time step
my_data_mean_across_space = ds_calcs_across_space.get_calc("mean").isel(time=0)
# trigger computation
my_data_mean_across_space.load()
```

[9]:
```
<xarray.DataArray 'TS' ()>
array(284.48966513)
Coordinates:
    time        object 1920-01-01 00:00:00
    collection  <U8 'orig'
Attributes:
    units:          K
    long_name:      Surface temperature (radiative)
```

(continues on next page)

(continued from previous page)

```
    cell_methods:   time: mean
    data_type:      cam-fv
    set_name:       orig
    cell_measures:  area: cell_area
```

[10]:
```
# Here just ask for the spatial mean at the first time step
d = ds_calcs_across_space.get_single_calc("most_repeated_percent")
# trigger computation
d
```

[10]:
```
4.159432870370371e-06
```

There are many currently available quantities to choose from. A complete list of derived quantities that can be passed to get_calc() is available here.

Spatial Plots

A Basic Spatial Plot

First we demonstrate the most basic usage of the ldcpy.plot() function. In its simplist form, ldcpy.plot() plots a single spatial plot of a dataset (i.e., no comparison) and requires:

- *ds* - the collection of datasets read in from ldcpy.open_datasets() (or ldcpy.collect_datasets())
- *varname* - the variable name we want to get statistics for
- *sets* - a list of the labels of the datasets in the collection that we are interested in
- *calc* - the name of the calculation to be plotted

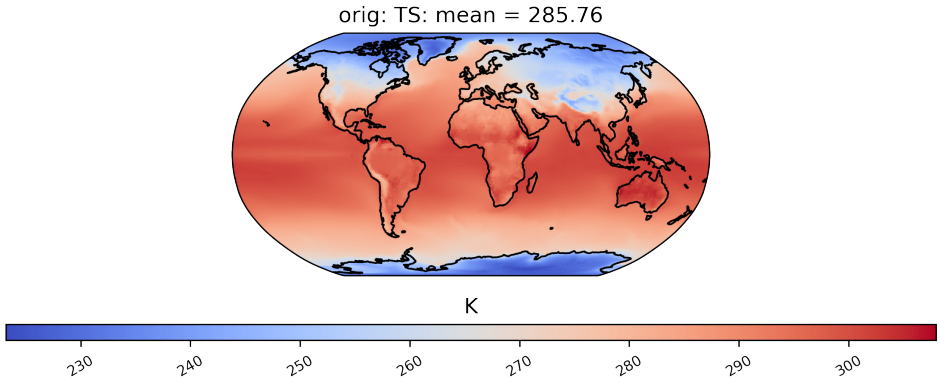
There are a number of optional arguments as well, that will be demonstrated in subsequent plots. These options include:

- *group_by*
- *scale*
- *calc_type*
- *plot_type*
- *transform*
- *subset*
- *lat*
- *lon*
- *lev*
- *color*
- *quantile*
- *start*
- *end*

A full explanation of each optional argument is available in the documentation here - as well as all available options. By default, a spatial plot of this data is created. The title of the plot contains the name of the dataset, the variable of interest, and the calculation that is plotted.

The following command generates a plot showing the mean TS (surface temperature) value from our dataset collection *col_ts* over time at each point in the dataset labeled 'orig'. Note that plots can be saved by using the matplotlib function savefig:

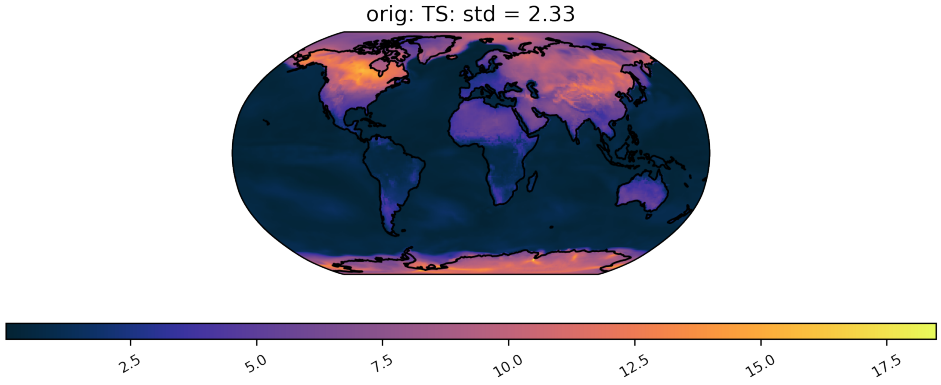
[11]:
```
ldcpy.plot(
    col_ts,
    "TS",
    sets=["orig"],
    calc="mean",
    plot_type="spatial",
)
```



nbsphinx-code-borderwhite

We can also plot quantities other than the mean, such as the standard deviation at each grid point over time. We can also change the color scheme (for a full list of quantitiess and color schemes, see the documentation). Here is an example of a plot of the same dataset from *col_ts* as above but using a different color scheme and the standard deviation calculation. Notice that we do not specify the 'plot_type' this time, because it defaults to 'spatial':

[12]:
```
# plot of the standard deviation of TS values in the col_ds "orig" dataset
ldcpy.plot(col_ts, "TS", sets=["orig"], calc="std", color="cmo.thermal")
```

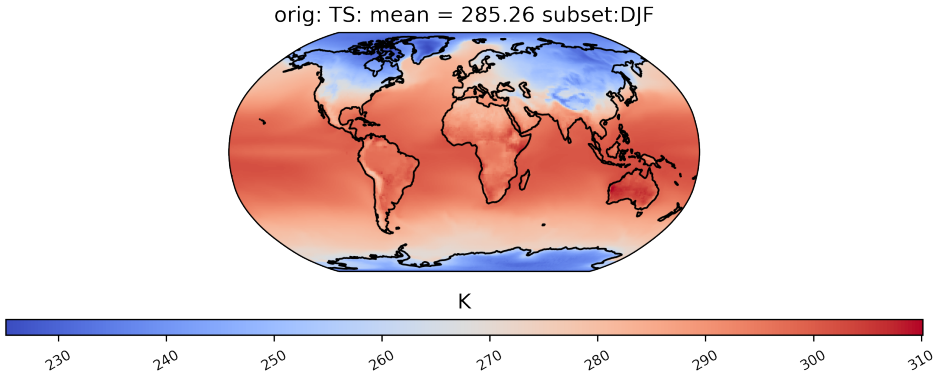


nbsphinx-code-borderwhite

Spatial Subsetting

Plotting a derived quantity of a subset of the data (e.g., not all of the time slices in the dataset) is possible using the subset keyword. In the plot below, we just look at "DJF" data (Dec., Jan., Feb.). Other options for subset are available here.

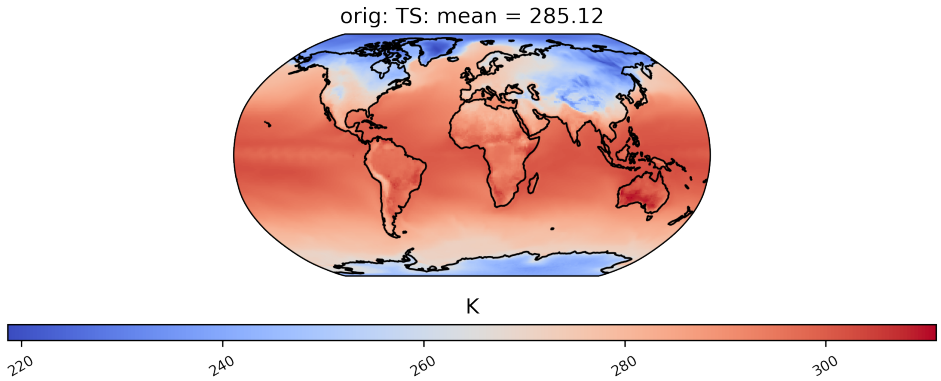
[13]:
```
# plot of mean winter TS values in the col_ts "orig" dataset
ldcpy.plot(col_ts, "TS", sets=["orig"], calc="mean", subset="DJF")
```



nbsphinx-code-borderwhite

It is also possible to plot calculations for a subset of the time slices by specifying the start and end indices of the time we are interested in. This command creates a spatial plot of the mean TS values in "orig" for the first five days of data:

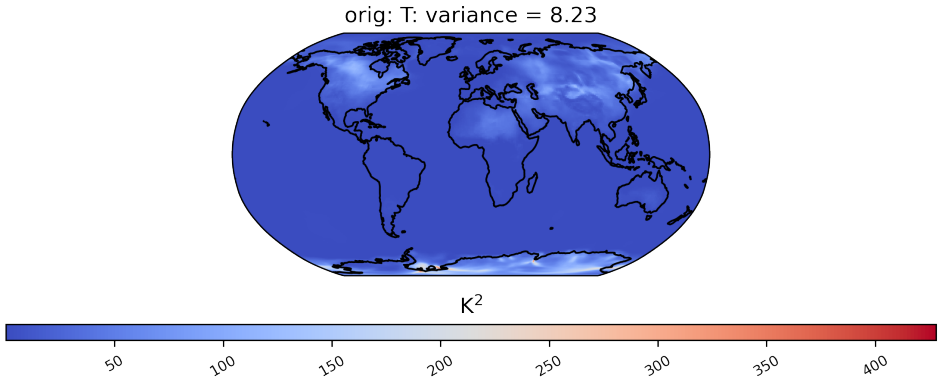
[14]:
```
# plot of the first 5 TS values in the ds orig dataset
ldcpy.plot(col_ts, "TS", sets=["orig"], calc="mean", start=0, end=5)
```



nbsphinx-code-borderwhite

Finally, for a 3D dataset, we can specify which vertical level to view using the "lev" keyword. Note that "lev" is a dimension in our dataset *col_t* (see printed output for *col_t* above), and in this case lev=30, meaning that lev ranges from 0 and 29, where 0 is at the surface (by default, lev=0):

[15]:
```
# plot of T values at lev=29 in the col_t "orig" dataset
ldcpy.plot(col_t, "T", sets=["orig"], calc="variance", lev=29)
```

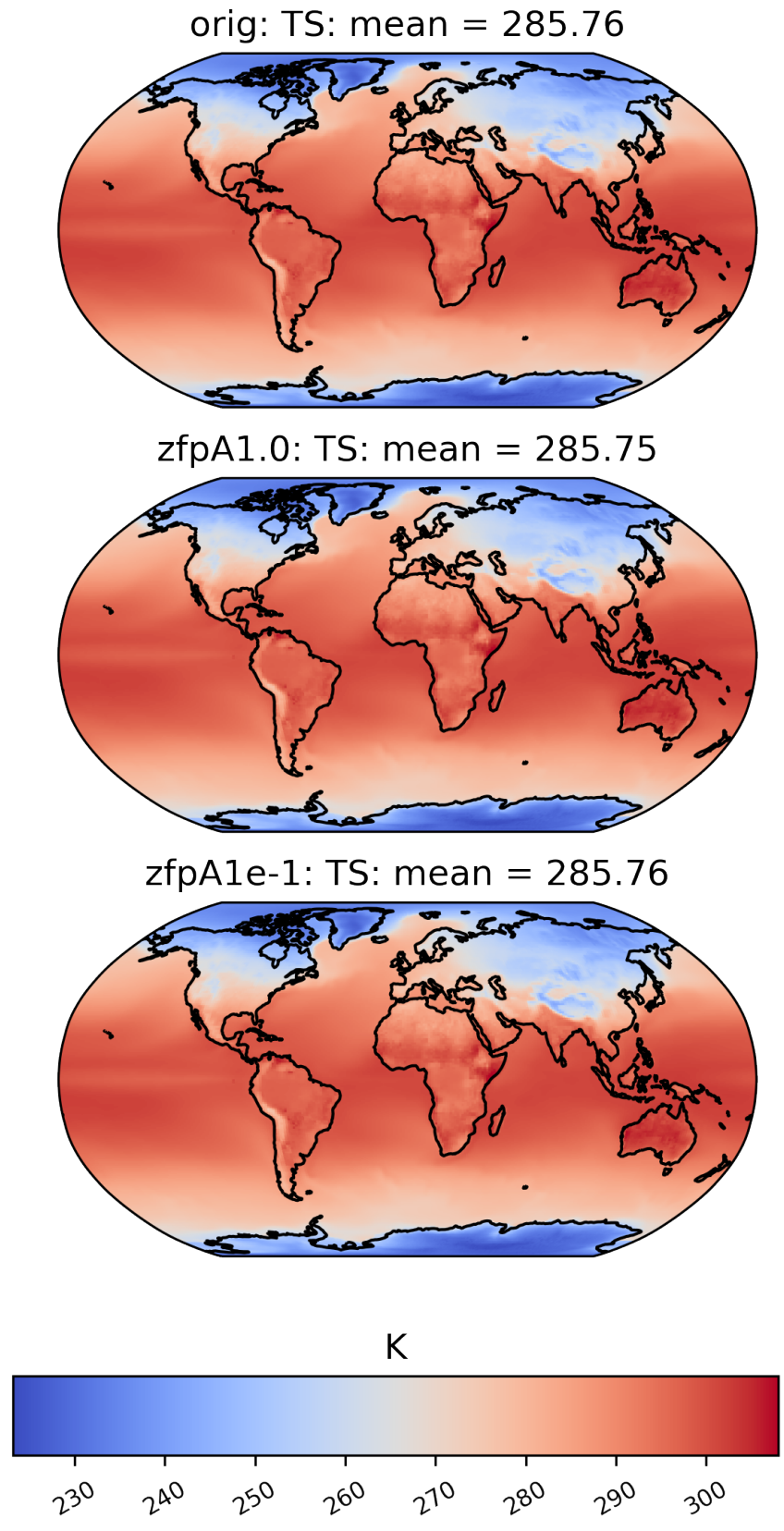


nbsphinx-code-borderwhite

Spatial Comparisons and Diff Plots

If we want a side-by-side comparison of two datasets, we need to specify an additional dataset name in the sets argument. The plot below shows the mean TS value over time at each point in the 'orig' (original), 'zfpA1.0' (compressed with zfp, absolute error tolerance 1.0) and 'zfpA1e-1' (compressed with zfp, absolute error tolerance 0.1) datasets in collection *col_ts* . The "vert_plot" argument inidcates that the arrangent of the subplots should be vertical.

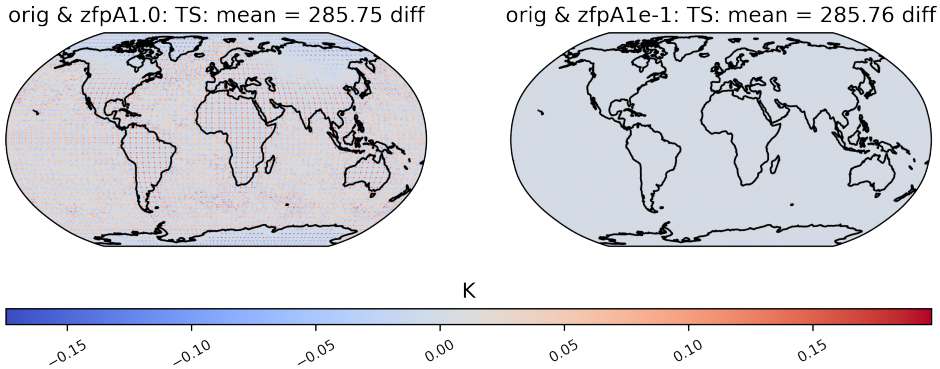
[16]:
```
# comparison between mean TS values in col_ts for "orig" and "zfpA1.0" datasets
ldcpy.plot(
    col_ts,
    "TS",
    sets=["orig", "zfpA1.0", "zfpA1e-1"],
    calc="mean",
    vert_plot=True,
)
```



nbsphinx-code-borderwhite

To the eye, these plots look identical. This is because the effects of compression are small compared to the magnitude of the data. We can view the compression artifacts more clearly by plotting the difference between two plots. This can be done by setting the calc_type to 'diff':

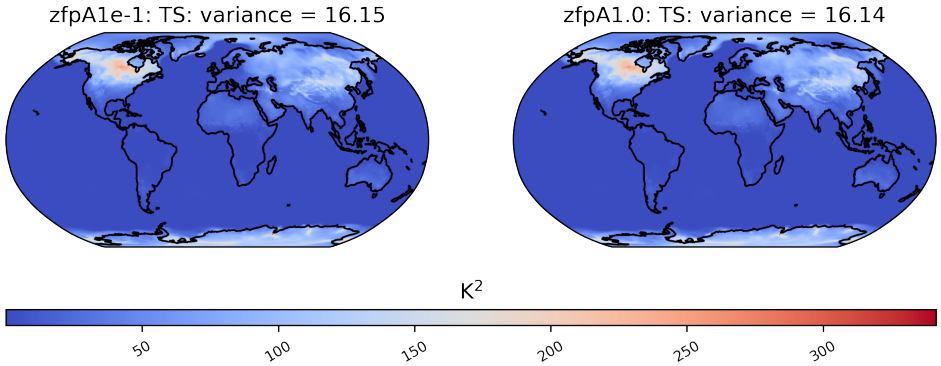
[17]:
```
# diff between mean TS values in col_ds "orig" and "zfpA1.0" datasets
ldcpy.plot(
    col_ts,
    "TS",
    sets=["orig", "zfpA1.0", "zfpA1e-1"],
    calc="mean",
    calc_type="diff",
)
```



nbsphinx-code-borderwhite

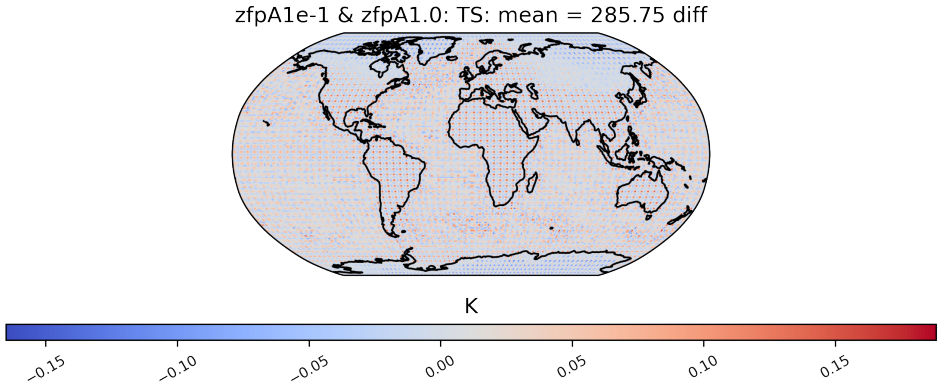
We are not limited to comparing side-by-side plots of the original and compressed data. It is also possible to compare two different compressed datasets side-by-side as well, by using a different dataset name for the first element in sets:

[18]:
```
# comparison between variance of TS values in col_ts for "zfpA1e-1" and "zfpA1.0 datasets
↪"
ldcpy.plot(col_ts, "TS", sets=["zfpA1e-1", "zfpA1.0"], calc="variance")
```



nbsphinx-code-borderwhite

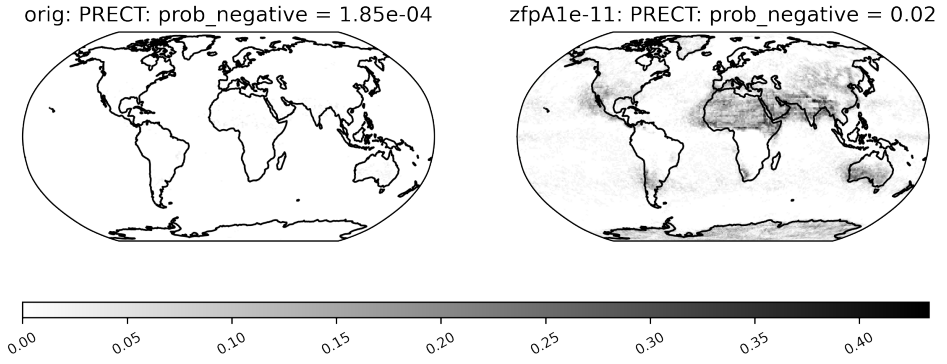
[19]:
```
# diff between mean TS values in col_ts for "zfpA1e-1" and "zfpA1.0" datasets
ldcpy.plot(
    col_ts,
    "TS",
    sets=["zfpA1e-1", "zfpA1.0"],
    calc="mean",
    calc_type="diff",
)
```



nbsphinx-code-borderwhite

Sometimes comparison plots can look strikingly different, indicating a potential problem with the compression. This plot shows the probability of negative rainfall (PRECT). We would expect this quantity to be zero everywhere on the globe (because negative rainfall does not make sense!), but the compressed output shows regions where the probability is significantly higher than zero:

[20]:
```
ldcpy.plot(
    col_prect,
    "PRECT",
    sets=["orig", "zfpA1e-11"],
    calc="prob_negative",
    color="binary",
)
```



nbsphinx-code-borderwhite

Unusual Values (inf, -inf, NaN)

Some derived quantities result in values that are +/- infinity, or NaN (likely resulting from operations like 0/0 or inf/inf). NaN values are plotted in neon green, infinity is plotted in white, and negative infinity is plotted in black (regardless of color scheme). If infinite values are present in the plot data, arrows on either side of the colorbar are shown to indicate the color for +/- infinity. This plot shows the log of the ratio of the odds of positive rainfall over time in the compressed and original output, log(odds_positive compressed/odds_positive original). Here we are suppressing all of the divide by zero warnings for aesthetic reasons.

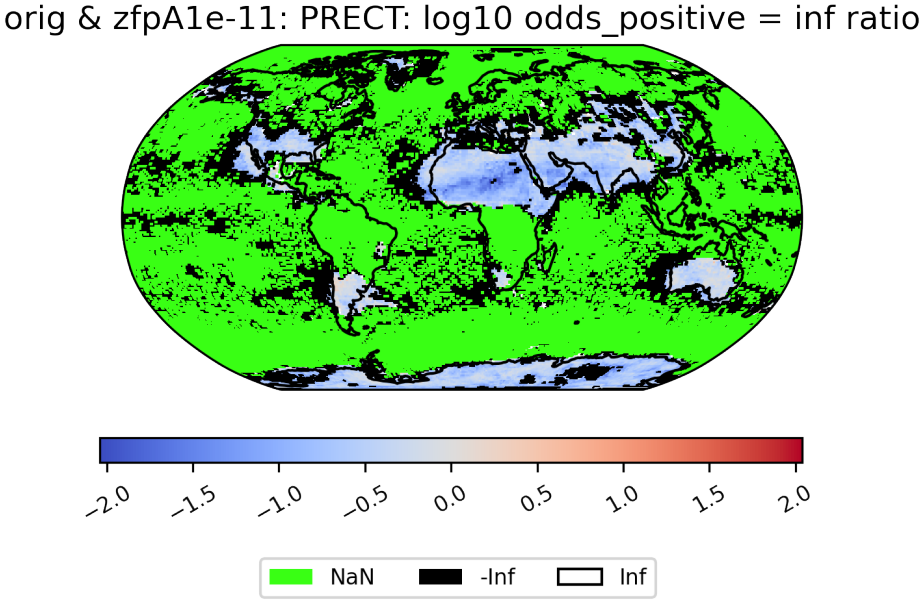
The following plot showcases some interesting plot features. We can see sections of the map that take NaN values, and other sections that are black because taking the log transform has resulted in many points with the value -inf:

[21]:
```
# plot of the ratio of odds positive PRECT values in col_prect zfpA1e-11 dataset / col_
↪prect orig dataset (log transform)
```

(continues on next page)

(continued from previous page)

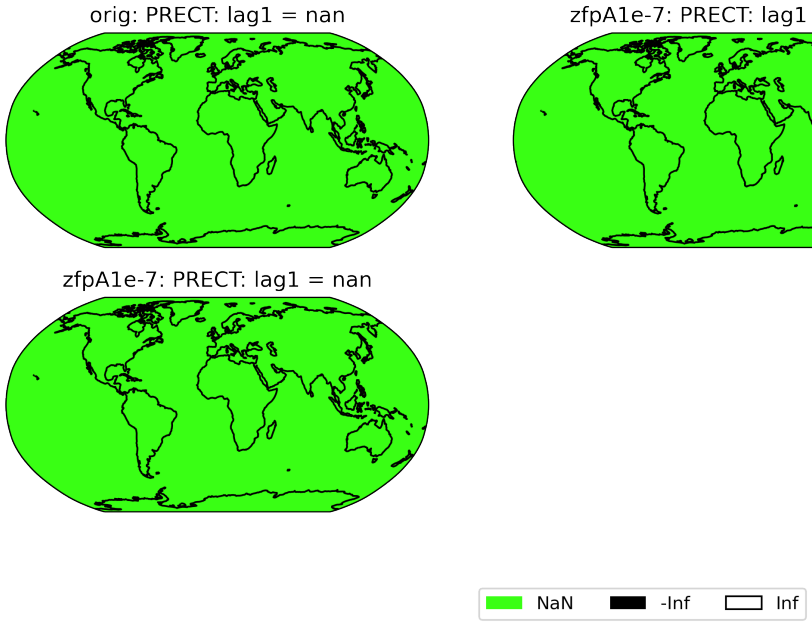
```
ldcpy.plot(
    col_prect,
    "PRECT",
    sets=["orig", "zfpA1e-11"],
    calc="odds_positive",
    calc_type="ratio",
    transform="log",
    axes_symmetric=True,
    vert_plot=True,
)
```



nbsphinx-code-borderwhite

If all values are NaN, then the colorbar is not shown. Instead, a legend is shown indicating the green(!) color of NaN values, and the whole plot is colored gray. (If all values are infinite, then the plot is displayed normally with all values either black or white). Because the example dataset only contains 60 days of data, the deseasonalized lag-1 values and their variances are all 0, and so calculating the correlation of the lag-1 values will involve computing 0/0 = NaN:

[22]:
```
# plot of lag-1 correlation of PRECT values in col_prect orig dataset
ldcpy.plot(col_prect, "PRECT", sets=["orig", "zfpA1e-7", "zfpA1e-7"], calc="lag1")
```

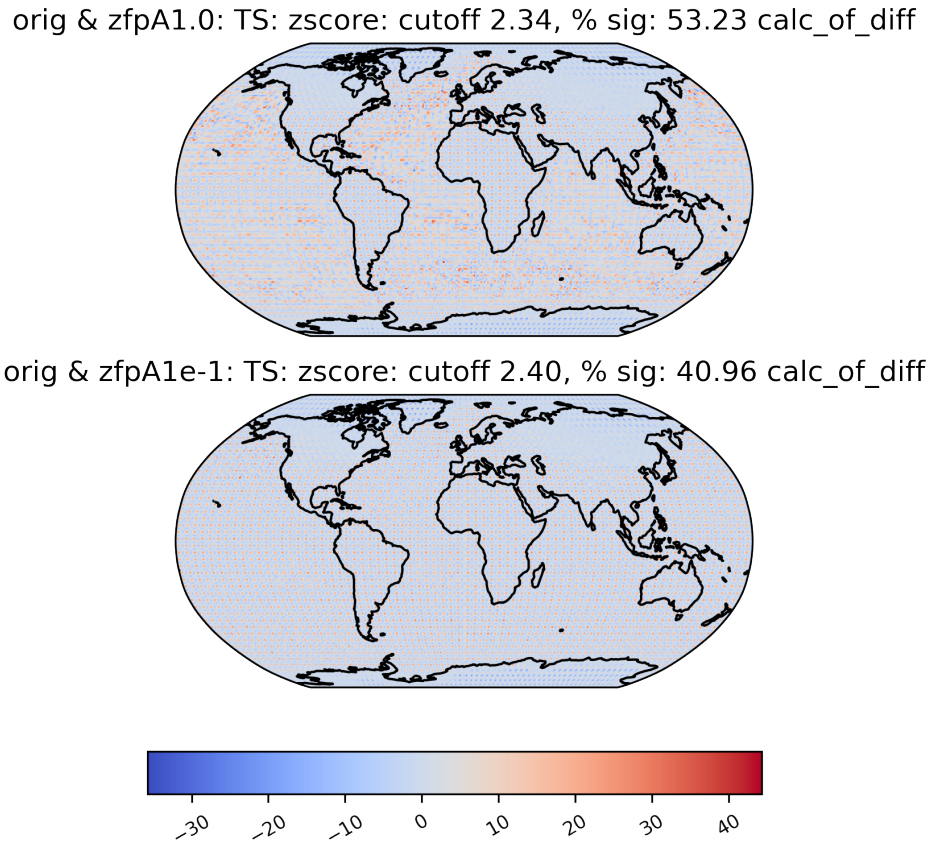


nbsphinx-code-borderwhite

Other Spatial Plots

Sometimes, we may want to compute a quantity on the difference between two datasets. For instance, the zscore calculation calculates the zscore at each point under the null hypothesis that the true mean is zero, so using the "calc_of_diff" calc_type calculates the zscore of the diff between two datasets (to find the values that are significantly different between the two datasets). The zscore calculation in particular gives additional information about the percentage of significant gridpoints in the plot title:

[23]:
```
# plot of z-score under null hypothesis that "orig" value= "zfpA1.0" value
ldcpy.plot(
    col_ts,
    "TS",
    sets=["orig", "zfpA1.0", "zfpA1e-1"],
    calc="zscore",
    calc_type="calc_of_diff",
    vert_plot=True,
)
```



nbsphinx-code-borderwhite

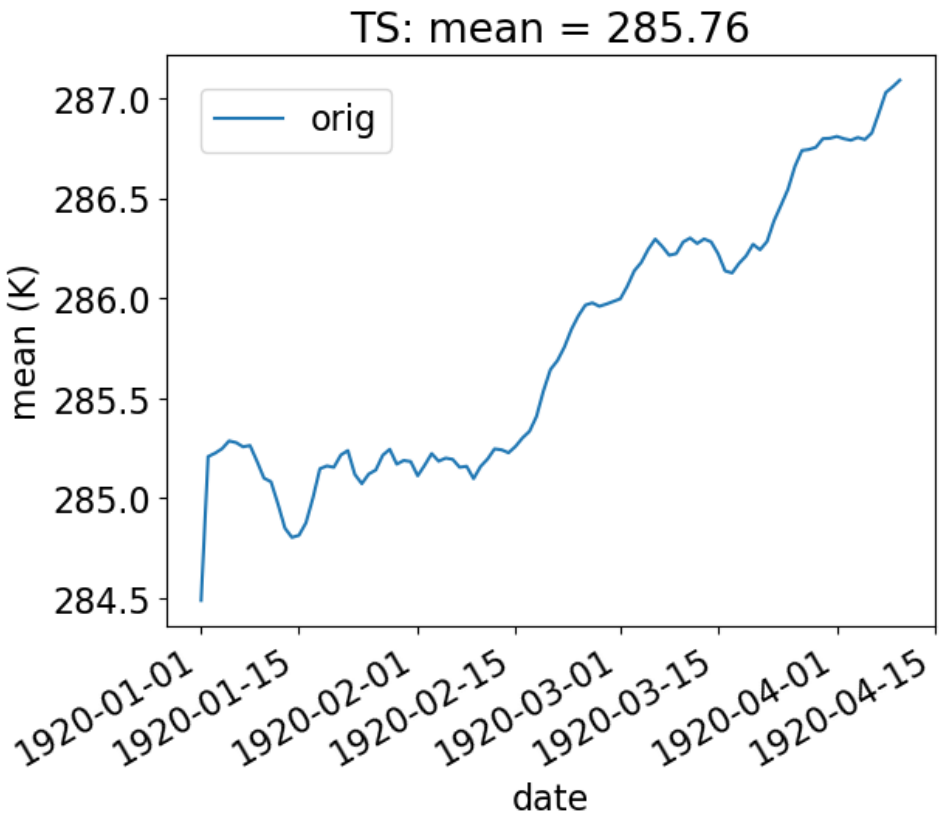
Time-Series Plots

We may want to aggregate the data spatially and look for trends over time. Therefore, we can also create a time-series plot of the calculations by changing the plot_type to "time_series". For time-series plots, the quantity values are on the y-axis and the x-axis represents time. We are also able to group the data by time, as shown below.

Basic Time-Series Plot

In the example below, we look at the 'orig' TS data in collection *col_ts* , and display the spatial mean at each day of the year (our data consists of 100 days).

[24]:
```
# Time-series plot of TS mean in col_ts orig dataset
ldcpy.plot(
    col_ts,
    "TS",
    sets=["orig"],
    calc="mean",
    plot_type="time_series",
    vert_plot=True,
    legend_loc="best",
)
```

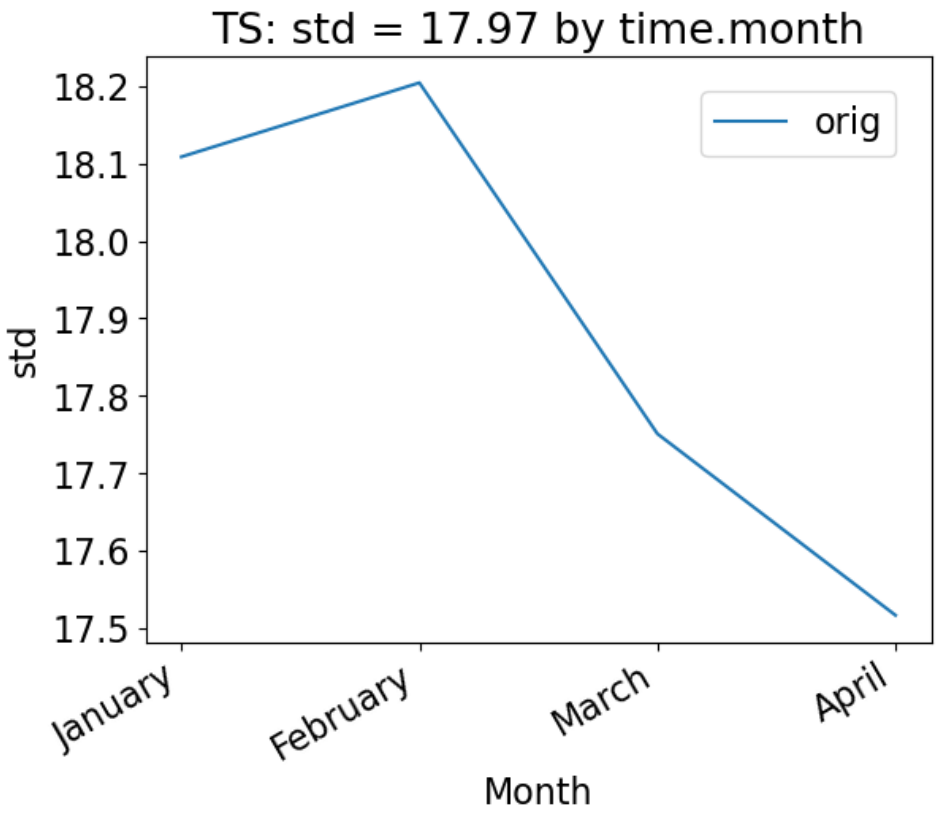


nbsphinx-code-borderwhite

Using the group_by keyword

To group the data by time, use the "group_by" keyword. This plot shows the mean standard deviation over all latitude and longitude points for each month. Note that this plot is not too interesting for our sample data, which has only 100 days of data.

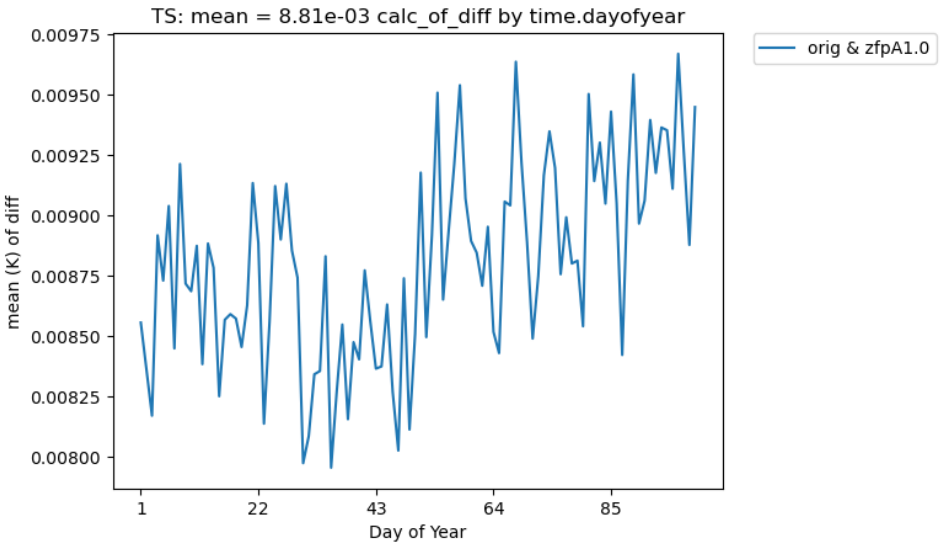
[25]:
```
# Time-series plot of TS standard deviation in col_ds "orig" dataset, grouped by month
ldcpy.plot(
    col_ts,
    "TS",
    sets=["orig"],
    calc="std",
    plot_type="time_series",
    group_by="time.month",
    vert_plot=True,
)
```



nbsphinx-code-borderwhite

One could also group by days of the year, as below. Again, because we have less than a year of data, this plot looks the same as the previous version. However, this approach would be useful with data containing multiple years.

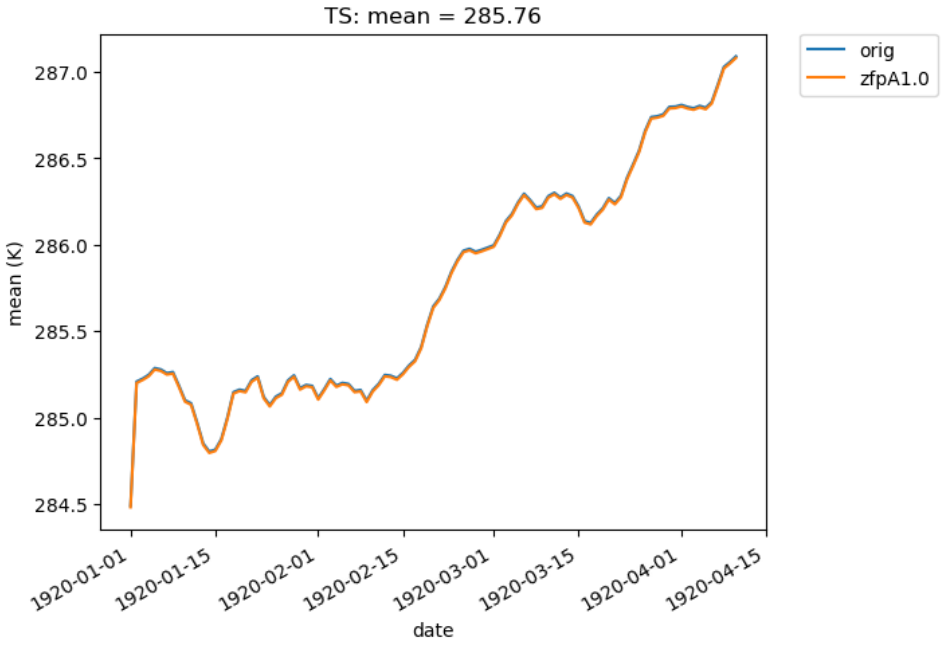
[26]:
```
# Time-series plot of TS mean in col_ts "orig" dataset, grouped by day of year
ldcpy.plot(
    col_ts,
    "TS",
    sets=["orig", "zfpA1.0"],
    calc="mean",
    calc_type="calc_of_diff",
    plot_type="time_series",
    group_by="time.dayofyear",
)
```



nbsphinx-code-borderwhite

We can also overlay multiple sets of time-series data on the same plot. For instance, we can plot the mean of two datasets over time. Note that the blue and orange lines overlap almost perfectly:

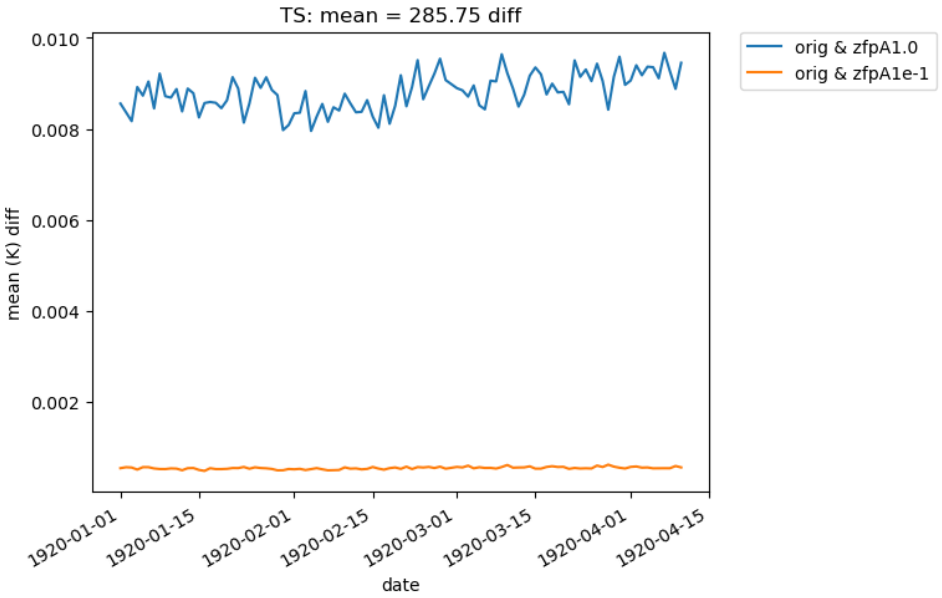
[27]:
```
# Time-series plot of TS mean in col_ts 'orig' dataset
ldcpy.plot(
    col_ts,
    "TS",
    sets=["orig", "zfpA1.0"],
    calc="mean",
    plot_type="time_series",
)
```



nbsphinx-code-borderwhite

If we change the calc_type to “diff”, “ratio” or “calc_of_diff”, the first element in sets is compared against subsequent elements in the sets list. For example, we can compare the difference in the mean of two compressed datasets to the original dataset like so:

[28]:
```
# Time-series plot of TS mean differences (with 'orig') in col_ts orig dataset
ldcpy.plot(
    col_ts,
    "TS",
    sets=["orig", "zfpA1.0", "zfpA1e-1"],
    calc="mean",
    plot_type="time_series",
    calc_type="diff",
)
```



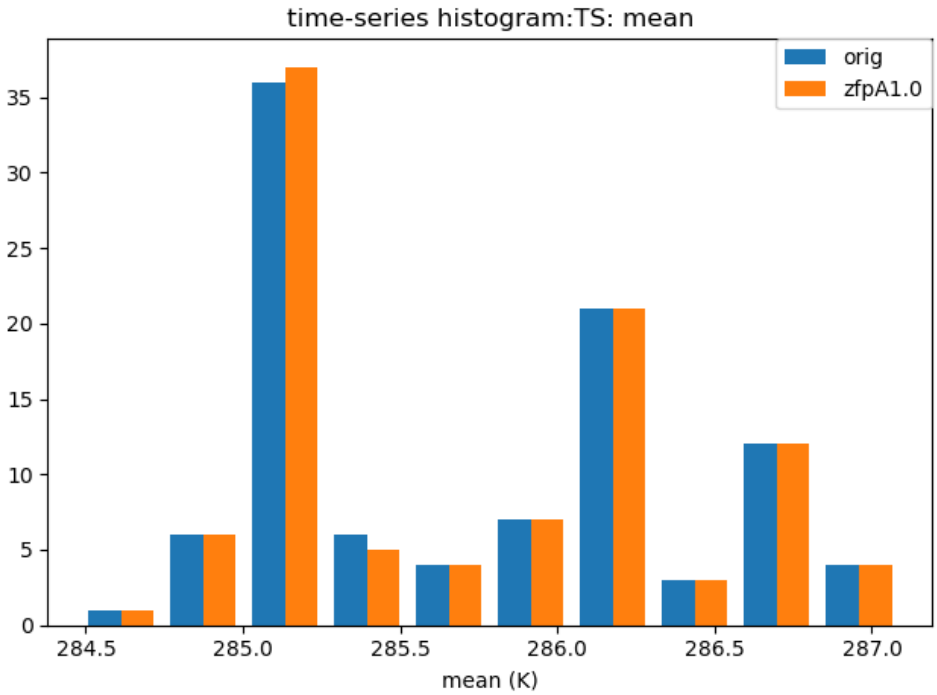
nbsphinx-code-borderwhite

Histograms

We can also create a histogram of the data by changing the plot_type to 'histogram'. Note that these histograms are currently generated from time-series quantity values (a histogram of spatial values is not currently available).

The histogram below shows the mean values of TS in the 'orig' and 'zfpA1.0' datasets in our collection *col_ts*. Recall that this dataset contains 100 timeslices.

[29]:
```
# Histogram of mean TS values in the col_ts (for'orig' and 'zfpA1.0') dataset

ldcpy.plot(col_ts, "TS", sets=["orig", "zfpA1.0"], calc="mean", plot_type="histogram")
```

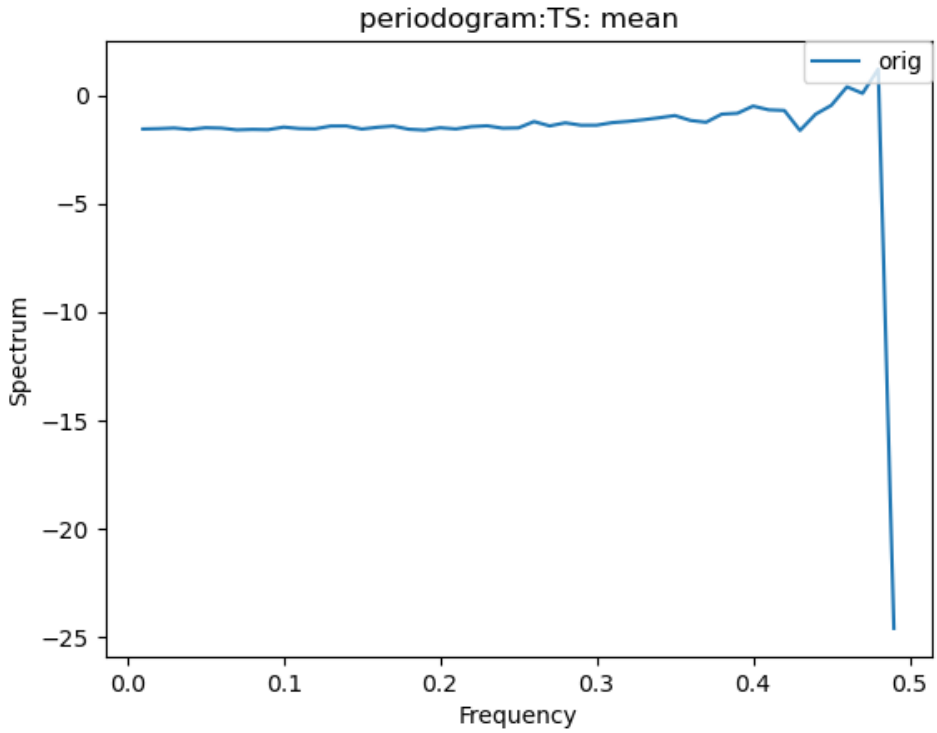


nbsphinx-code-borderwhite

Other Time-Series Plots

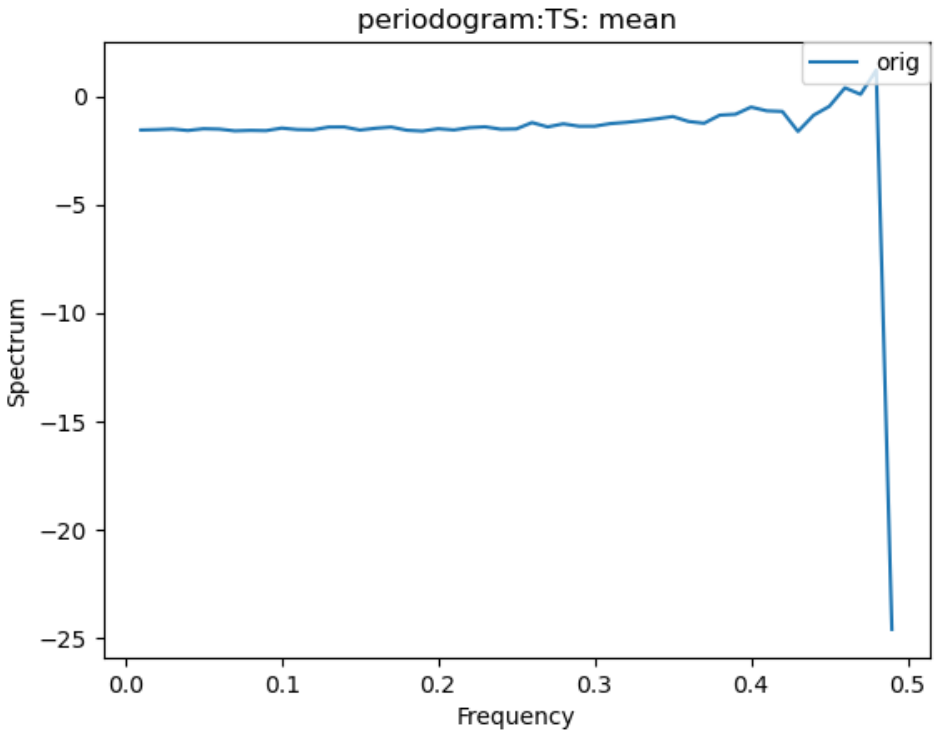
We can create a periodogram based on a dataset as well, by specifying a plot_type of "periodogram".

[30]:
```
ldcpy.plot(col_ts, "TS", sets=["orig"], calc="mean", plot_type="periodogram")
```



nbsphinx-code-borderwhite

[31]:
```
ldcpy.plot(col_ts, "TS", sets=["orig"], calc="mean", plot_type="periodogram")
```

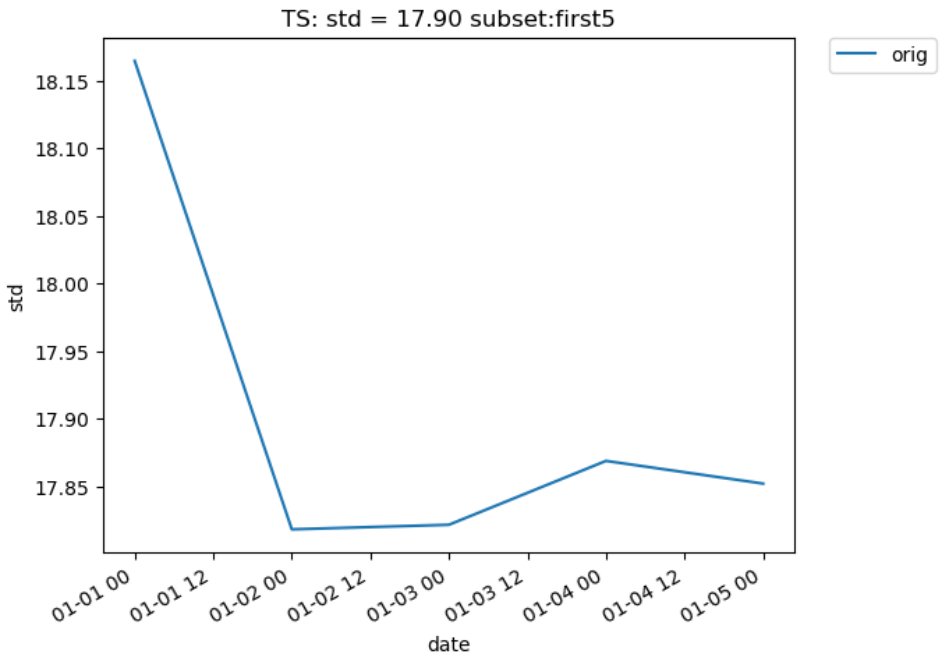


nbsphinx-code-borderwhite

Subsetting

Subsetting is also possible on time-series data. The following plot makes use of the subset argument, which is used to plot derived quantities on only a portion of the data. A full list of available subsets is available here. The following plot uses the 'first5' subset, which only shows the quantity values for the first five time slices (in this case, days) of data:

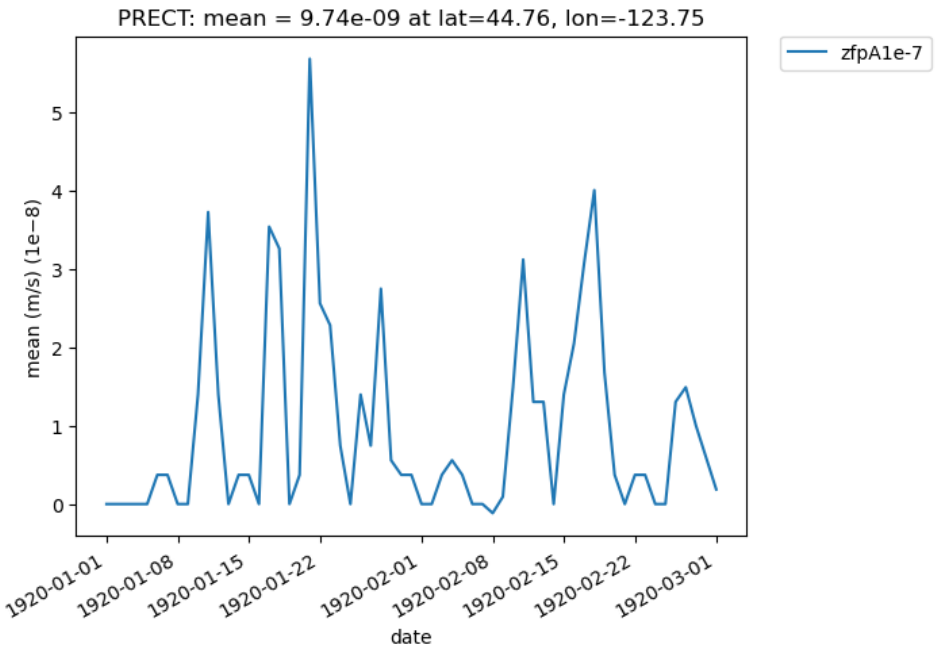
[32]:
```
# Time-series plot of first five TS standard deviations in col_ts "orig" dataset
ldcpy.plot(
    col_ts,
    "TS",
    sets=["orig"],
    calc="std",
    plot_type="time_series",
    subset="first5",
)
```



nbsphinx-code-borderwhite

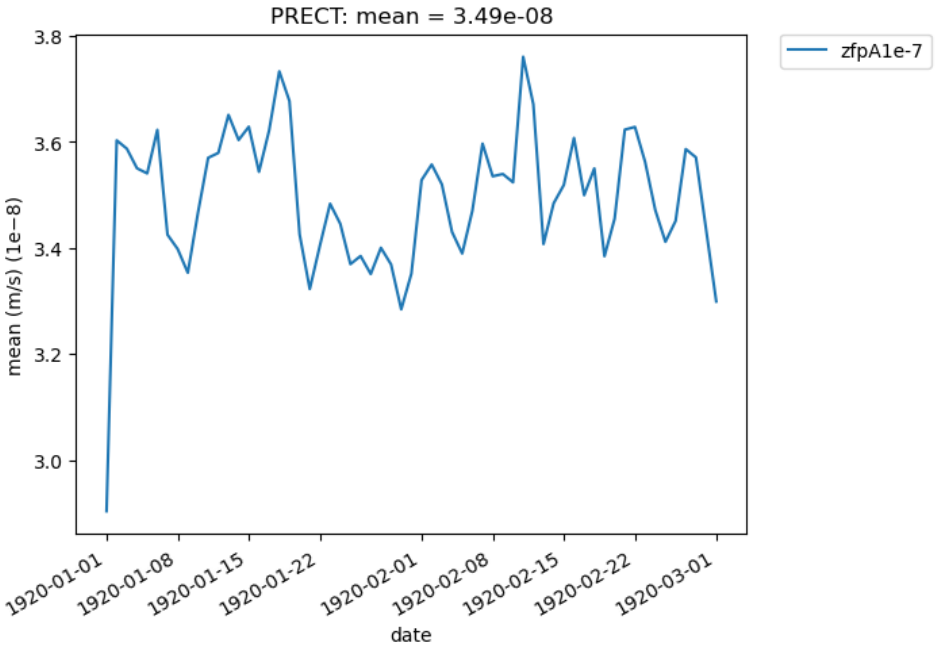
Additionally, we can specify "lat" and "lon" keywords for time-series plots that give us a subset of the data at a single point, rather than averaging over all latitudes and longitudes. The nearest latitude and longitude point to the one specified is plotted (and the actual coordinates of the point can be found in the plot title). This plot, for example, shows the difference in mean rainfall between the compressed and original data at the location (44.76, -123.75):

[33]:
```
# Time series plot of PRECT mean data for col_prect "zfpA1e-7" dataset at the location␣
↪(44.76, -123.75)

ldcpy.plot(
    col_prect,
    "PRECT",
    sets=["zfpA1e-7"],
    calc="mean",
    plot_type="time_series",
    lat=44.76,
    lon=-123.75,
)
```



nbsphinx-code-borderwhite

[34]:
```
# Time series plot of PRECT mean data for col_prect "zfpA1e-7" dataset at the location␣
↪(44.76, -123.75)

ldcpy.plot(col_prect, "PRECT", sets=["zfpA1e-7"], calc="mean", plot_type="time_series")
```



nbsphinx-code-borderwhite

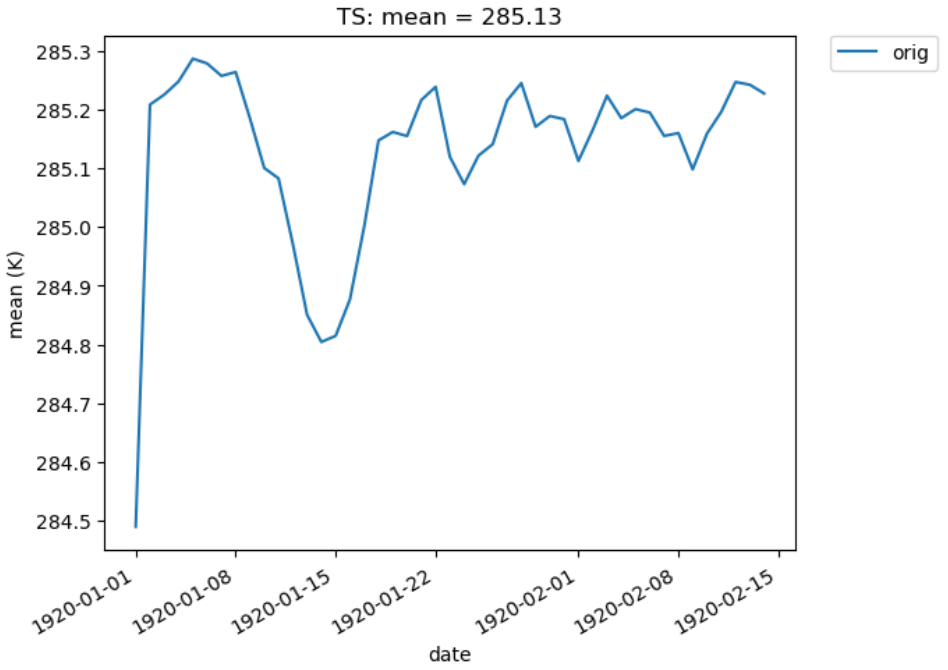
It is also possible to plot quantities for a subset of the data, by specifying the start and end indices of the data we are interested in. This command creates a time-series plot of the mean TS values for the first 45 days of data:

[35]:
```
# Time series plot of first 45 TS mean data points for col_ts "orig" dataset
```

(continues on next page)

(continued from previous page)

```
ldcpy.plot(
    col_ts,
    "TS",
    sets=["orig"],
    calc="mean",
    start=0,
    end=44,
    plot_type="time_series",
)
```



nbsphinx-code-borderwhite

[ ]:

[ ]:

2.4.2 CURRENTLY NOT WORKING....needs updating!

2.4.3 Using data from AWS

A significant amount of Earth System Model (ESM) data is publicly available online, including data from the CESM Large Ensemble, CMIP5, and CMIP6 datasets. For accessing a single file, we can specify the file (typically netcdf or zarr format) and its location and then use fsspec (the "Filesystem Spec+ python package) and xarray to create a array.dataset. For several files, the intake_esm python module (https://github.com/intake/intake-esm) is particularly nice for obtaining the data and put it into an xarray.dataset.

This notebook assumes familiarity with the Tutorial Notebook. It additionally shows how to gather data from an ESM collection, put it into a dataset, and then create simple plots using the data with ldcpy.

Example Data

The example data we use is from the CESM Large Ensemble, member 31. This ensemble data has been lossily compressed and reconstructed as part of a blind evaluation study of lossy data compression in LENS (e.g., http://www.cesm.ucar.edu/projects/community-projects/LENS/projects/lossy-data-compression.html or https://gmd.copernicus.org/articles/9/4381/2016/).

Most of the data from the CESM Large Ensemble Project has been made available on Amazon Web Services (Amazon S3), see http://ncar-aws-www.s3-website-us-west-2.amazonaws.com/CESM_LENS_on_AWS.htm .

For comparison purposes, the original (non-compressed) data for Ensemble 31 has recently been made available on Amazon Web Services (Amazon S3) in the "ncar-cesm-lens-baker-lossy-compression-test" bucket.

[ ]:
```
# Add ldcpy root to system path
import sys

sys.path.insert(0, '../../../')

# Import ldcpy package
# Autoreloads package everytime the package is called, so changes to code will be␣
↪reflected in the notebook if the above sys.path.insert(...) line is uncommented.
%load_ext autoreload
%autoreload 2
import fsspec
import intake
import xarray as xr

import ldcpy

# display the plots in this notebook
%matplotlib inline

# silence warnings
import warnings

warnings.filterwarnings("ignore")
```

First, specify the filesystem and location of the data. Here we are accessing the original data from CESM-LENS ensemble 31, which is available on Amazon S3 in the store named *"ncar-cesm-lens-baker-lossy-compression-test"* bucket.

First we listing all available files (which are timeseries files containing a single variable) for that dataset. Note that unlike in the TutorialNotebook (which used NetCDF files), these files are all zarr format. Both monthly and daily data is available.

[ ]:
```
fs = fsspec.filesystem("s3", anon=True)
stores = fs.ls("ncar-cesm-lens-baker-lossy-compression-test/lens-ens31/")[1:]
stores[:]
```

Then we select the file from the store that we want and open it as an xarray.Dataset using xr.open_zarr(). Here we grab data for the first 2D daily variable, FLNS (net longwave flux at surface, in W/m^2), in the list (accessed by it location – stores[0]).

[ ]:
```
store = fs.get_mapper(stores[0])
ds_flns = xr.open_zarr(store, consolidated=True)
ds_flns
```

The above returned an xarray.Dataset.

Now let's grab the TMQ (Total vertically integrated precipitatable water) and the TS (surface temperature data) and PRECT (precipitation rate) data from AWS.

[ ]:
```
# TMQ data
store2 = fs.get_mapper(stores[16])
ds_tmq = xr.open_zarr(store2, consolidated=True)
# TS data
store3 = fs.get_mapper(stores[20])
ds_ts = xr.open_zarr(store3, consolidated=True)
# PRECT data
store4 = fs.get_mapper(stores[11])
ds_prect = xr.open_zarr(store4, consolidated=True)
```

Now we have the original data for FLNS and TMQ and TS and PRECT. Next we want to get the lossy compressed variants to compare with these.

Method 2: Using intake_esm

Now we will demonstrate using the intake_esm module to get the lossy variants of the files retrieved above. We can use the intake_esm module to search for and open several files as xarray.Dataset objects. The code below is modified from the intake_esm documentation, available here: https://intake-esm.readthedocs.io/en/latest/?badge=latest#overview.

We want to use ensemble 31 data from the CESM-LENS collection on AWS, which (as explained above) has been subjected to lossy compression. Many catalogs for publicly available datasets are accessible via intake-esm can be found at https://github.com/NCAR/intake-esm-datastore/tree/master/catalogs, including for CESM-LENS. We can open that collection as follows (see here: https://github.com/NCAR/esm-collection-spec/blob/master/collection-spec/collection-spec.md#attribute-object):

[ ]:
```
aws_loc = (
    "https://raw.githubusercontent.com/NCAR/cesm-lens-aws/master/intake-catalogs/aws-
↪cesm1-le.json"
)
aws_col = intake.open_esm_datastore(aws_loc)
aws_col
```

Next, we search for the subset of the collection (dataset and variables) that we are interested in. Let's grab FLNS, TMQ, and TS daily data from the atm component for our comparison (available data in this collection is listed here: http://ncar-aws-www.s3-website-us-west-2.amazonaws.com/CESM_LENS_on_AWS.htm).

[ ]:
```
# we want daily data for FLNS, TMQ, and TS and PRECT
aws_col_subset = aws_col.search(
    component="atm",
    frequency="daily",
    experiment="20C",
    variable=["FLNS", "TS", "TMQ", "PRECT"],
)
# display header info to verify that we got the right variables
aws_col_subset.df.head()
```

Then we load matching catalog entries into xarray datasets (https://intake-esm.readthedocs.io/en/latest/api.html#intake_esm.core.esm_datastore.to_dataset_dict). We create a dictionary of datasets:

```
[ ]: dset_dict = aws_col_subset.to_dataset_dict(
         zarr_kwargs={"consolidated": True, "decode_times": True},
         storage_options={"anon": True},
         cdf_kwargs={"chunks": {}, "decode_times": False},
     )
     dset_dict
```

Check the dataset keys to ensure that what we want is present. Here we only have oneentry in the dictonary as we requested the same time period and output frequency for all variables:

```
[ ]: dset_dict.keys()
```

Finally, put the dataset that we are interested from the dictionary into its own dataset variable. (We want the 20th century daily data – which is our only option.)

Also note from above that there are 40 ensemble members - and we just want ensemble 31 (member_id = 30 as can be seen in the coordinates above).

```
[ ]: aws_ds = dset_dict["atm.20C.daily"]
     aws_ds = aws_ds.isel(member_id=30)
     aws_ds
```

Now we have datasets for the original and the lossy compressed data for FLNS, TMQ, PRECT, and TS, which we can extract into a dataset for each variable:

```
[ ]: # extract the three variables from aws_ds as datasets
     aws_flns = aws_ds["FLNS"].to_dataset()
     aws_tmq = aws_ds["TMQ"].to_dataset()
     aws_ts = aws_ds["TS"].to_dataset()
     aws_prect = aws_ds["PRECT"].to_dataset()
```

2.4.4 Use *ldcpy* to compare the original data to the lossy compressed data

To use ldcpy, we need to group the data that we want to compare (like variables) into dataset collections. In the Tutorial Notebook, we used ldcpy.open_datasets() to do this as we needed to get the data from the NetCDF files. Here we already loaded the data from AWS into datasets, so we just need to use ldcpy.collect_datasets() to form collections of the datasets that we want to compare.

ldcpy.collect_datasets() requires the following three arguments:

- *varnames* : the variable(s) of interest to combine across files (typically the timeseries file variable name)
- *list_of_ds* : a list of the xarray datasets
- *labels* : a corresponding list of names (or labels) for each dataset in the collection

Note: This function is a wrapper for xarray.concat(), and any additional key/value pairs passed in as a dictionary are used as arguments to xarray.concat().

We will create 4 collections for ldcpy (one each for FLNS, TMQ, PRECT, and TS) and assign labels “original” and “lossy” to the respective datasets.

```
[ ]: # FLNS collection
     col_flns = ldcpy.collect_datasets("cam-fv", ["FLNS"], [ds_flns, aws_flns], ["original",
     ↪"lossy"])
     col_flns
```

```
[ ]: # TMQ collection
col_tmq = ldcpy.collect_datasets("cam-fv", ["TMQ"], [ds_tmq, aws_tmq], ["original",
↪"lossy"])
```

```
[ ]: # Ts collection
col_ts = ldcpy.collect_datasets("cam-fv", ["TS"], [ds_ts, aws_ts], ["original", "lossy"])
```

```
[ ]: # PRECT collection
col_prect = ldcpy.collect_datasets(
    "cam-fv", ["PRECT"], [ds_prect, aws_prect], ["original", "lossy"]
)
```

Now that we have our collections, we can do some comparisons. Note that these are large files, so make sure you have sufficient compute/memory.

```
[ ]: # Time-series plot of PRECT mean in col_ds 'original' dataset - first 100 daysa
ldcpy.plot(
    col_prect,
    "PRECT",
    sets=["original", "lossy"],
    calc="mean",
    plot_type="time_series",
    start=0,
    end=100,
)
```

```
[ ]: # print statistics about 'original', 'lossy', and diff between the two datasets for TMQ at␣
↪time slice 365
ldcpy.compare_stats(col_tmq.isel(time=365), "TMQ", ["original", "lossy"])
```

The original data for FLNS and TMQ and TS and PRECT (from Amazon S3 in the "ncar-cesm-lens-baker-lossy-compression-test" bucket) was loaded above using method 1. An alternative would be to create our own catalog for this data for use with intake-esm. To illustrate this, we created a test_catalog.csv and test_collection.json file for this particular simple example.

We first open our collection.

```
[ ]: my_col_loc = "./collections/test_collection.json"
my_col = intake.open_esm_datastore(my_col_loc)
my_col
```

```
[ ]: # printing the head() gives us the file names
my_col.df.head()
```

Let's load all of these into our dictionary! (So we don't need to do the search to make a subset of variables as above in Method 2.)

```
[ ]: my_dset_dict = my_col.to_dataset_dict(
    zarr_kwargs={"consolidated": True, "decode_times": True},
    storage_options={"anon": True},
)
my_dset_dict
```

```
[ ]: # we again just want the 20th century daily data
     my_ds = my_dset_dict["atm.20C.daily"]
     my_ds
```

Now we can make a dataset for each variable as before.

```
[ ]: my_ts = my_ds["TS"].to_dataset()
     my_tmq = my_ds["TMQ"].to_dataset()
     my_prect = my_ds["PRECT"].to_dataset()
     my_flns = my_ds["FLNS"].to_dataset()
```

And now we can form new collections as before and do comparisons…

```
[ ]:
```

2.4.5 NCAR JupyterHub Large Data Example Notebook

Note: If you do not have access to the NCAR machine, please look at the AWS-LENS example notebook instead.

This notebook demonstrates how to compare large datasets on glade with ldcpy. In particular, we will look at data from CESM-LENS1 project (http://www.cesm.ucar.edu/projects/community-projects/LENS/data-sets.html). In doing so, we will start a DASK client from Jupyter. This notebook is meant to be run on NCAR's JupyterHub (https://jupyterhub.hpc.ucar.edu). We will use a subset of the CESM-LENS1 data on glade is located in /glade/p/cisl/asap/ldcpy_sample_data/lens.

We assume that you have a copy of the ldcpy code on NCAR's glade filesystem, obtained via: `git clone https://github.com/NCAR/ldcpy.git`

When you launch your NCAR JupyterHub session, you will need to indicate a machine and then you will need your charge account. You can then launch the session and navigate to this notebook.

NCAR's JupyterHub documentation: https://www2.cisl.ucar.edu/resources/jupyterhub-ncar

Here's another good resource for using NCAR's JupyterHub: https://ncar-hackathons.github.io/jupyterlab-tutorial/jhub.html)

You need to run your notebook with the "cmip6-2019.10" kernel (choose from the dropdown in the upper left.)

Note that the compressed data that we are using was generated for this paper:

Allison H. Baker, Dorit M. Hammerling, Sheri A. Mickelson, Haiying Xu, Martin B. Stolpe, Phillipe Naveau, Ben Sanderson, Imme Ebert-Uphoff, Savini Samarasinghe, Francesco De Simone, Francesco Carbone, Christian N. Gencarelli, John M. Dennis, Jennifer E. Kay, and Peter Lindstrom, "Evaluating Lossy Data Compression on Climate Simulation Data within a Large Ensemble." Geoscientific Model Development, 9, pp. 4381-4403, 2016 (https://gmd.copernicus.org/articles/9/4381/2016/)

Setup

Let's set up our environment. First, make sure that you are using the cmip6-2019.10 kernel. Then you will need to modify the path below to indicate where you have cloned ldcpy. (*Note: soon we will install ldcpy on Cheyenne/Casper in the cmpi6-2019.10 kernel* .)

If you want to use the dask dashboard, then the dask.config link must be set below (modify for your path in your browser).

[1]:
```
# Make sure you are using the cmpi6-2019.10 kernel on NCAR JupyterHub

# Add ldcpy root to system path (MODIFY FOR YOUR LDCPY CODE LOCATION)
import sys

sys.path.insert(0, '/glade/u/home/abaker/repos/ldcpy')
import ldcpy

# Display output of plots directly in Notebook
%matplotlib inline

# Automatically reload module if it is editted
%reload_ext autoreload
%autoreload 2

# silence warnings
import warnings

warnings.filterwarnings("ignore")


# if you want to use the DASK daskboard, then modify the below and run
# import dask
# dask.config.set(
#    {'distributed.dashboard.link': 'https://jupyterhub.hpc.ucar.edu/ch/user/abaker/proxy/
↪{port}/status'}
# )
```

Connect to DASK distributed cluster :

For NCAR machines, we can use ncar_jobqueue package (instead of the dask_jobqueue package). (The cluster object is for a single compute node: https://jobqueue.dask.org/en/latest/howitworks.html)

[2]:
```
from dask.distributed import Client
from ncar_jobqueue import NCARCluster

cluster = NCARCluster(project='NTDD0004')
# scale as needed
cluster.adapt(minimum_jobs=1, maximum_jobs=30)
cluster
client = Client(cluster)
```

The scheduler creates a normal-looking job script that it can submit multiple times to the queue:

[3]:
```
# Look at the job script (optional)
print(cluster.job_script())
```

```
#!/usr/bin/env bash

#PBS -N dask-worker-cheyenne
#PBS -q regular
#PBS -A NTDD0004
#PBS -l select=1:ncpus=36:mem=109GB
#PBS -l walltime=01:00:00
#PBS -e /glade/scratch/abaker/dask/cheyenne/logs/
#PBS -o /glade/scratch/abaker/dask/cheyenne/logs/

/ncar/usr/jupyterhub/envs/cmip6-201910/bin/python -m distributed.cli.dask_worker tcp://
↪10.148.1.55:42017 --nthreads 1 --nprocs 18 --memory-limit 5.64GiB --name dummy-name --
↪nanny --death-timeout 60 --local-directory /glade/scratch/abaker/dask/cheyenne/local-
↪dir --interface ib0 --protocol tcp://
```

The sample data on the glade filesystem

In /glade/p/cisl/asap/ldcpy_sample_data/lens on glade, we have TS (surface temperature), PRECT (precipiation rate), and PS (surface pressure) data from CESM-LENS1. These all all 2D variables. TS and PRECT have daily output, and PS has monthly output. We have the compressed and original versions of all these variables that we would like to compare with ldcpy.

First we list what is in this directory (two subdirectories):

[4]:
```
# list directory contents
import os

os.listdir("/glade/p/cisl/asap/ldcpy_sample_data/lens")
```

[4]:
```
['README.txt', 'lossy', 'orig']
```

Now we look at the contents of each subdirectory. We have 14 files in each, consisting of 2 different timeseries files for each variable (1920-2005 and 2006-2080).

[5]:
```
# list lossy directory contents (files that have been lossy compressed and reconstructed)
lossy_files = os.listdir("/glade/p/cisl/asap/ldcpy_sample_data/lens/lossy")
lossy_files
```

[5]:
```
['c.FLUT.daily.20060101-20801231.nc',
 'c.CCN3.monthly.192001-200512.nc',
 'c.FLNS.monthly.192001-200512.nc',
 'c.LHFLX.daily.20060101-20801231.nc',
 'c.TS.daily.19200101-20051231.nc',
 'c.SHFLX.daily.20060101-20801231.nc',
 'c.U.monthly.200601-208012.nc',
 'c.TREFHT.monthly.192001-200512.nc',
 'c.LHFLX.daily.19200101-20051231.nc',
 'c.TMQ.monthly.192001-200512.nc',
 'c.PRECT.daily.20060101-20801231.nc',
 'c.PS.monthly.200601-208012.nc',
```

(continues on next page)

(continued from previous page)

```
 'c.Z500.daily.20060101-20801231.nc',
 'c.FLNS.monthly.200601-208012.nc',
 'c.Q.monthly.192001-200512.nc',
 'c.U.monthly.192001-200512.nc',
 'c.CLOUD.monthly.192001-200512.nc',
 'c.so4_a2_SRF.daily.20060101-20801231.nc',
 'c.PRECT.daily.19200101-20051231.nc',
 'c.TAUX.daily.20060101-20801231.nc',
 'c.CLOUD.monthly.200601-208012.nc',
 'c.TS.daily.20060101-20801231.nc',
 'c.FSNTOA.daily.20060101-20801231.nc',
 'c.Q.monthly.200601-208012.nc',
 'c.CCN3.monthly.200601-208012.nc',
 'c.TREFHT.monthly.200601-208012.nc',
 'c.TMQ.monthly.200601-208012.nc',
 'c.PS.monthly.192001-200512.nc']
```

[6]:
```
# list orig (i.e., uncompressed) directory contents
orig_files = os.listdir("/glade/p/cisl/asap/ldcpy_sample_data/lens/orig")
orig_files
```

[6]:
```
['CLOUD.monthly.200601-208012.nc',
 'LHFLX.daily.20060101-20801231.nc',
 'so4_a2_SRF.daily.20060101-20801231.nc',
 'PRECT.daily.20060101-20801231.nc',
 'TMQ.monthly.192001-200512.nc',
 'TAUX.daily.20060101-20801231.nc',
 'CCN3.monthly.200601-208012.nc',
 'PS.monthly.192001-200512.nc',
 'FLUT.daily.20060101-20801231.nc',
 'PRECT.daily.19200101-20051231.nc',
 'TS.daily.20060101-20801231.nc',
 'PS.monthly.200601-208012.nc',
 'TS.daily.19200101-20051231.nc',
 'Z500.daily.20060101-20801231.nc',
 'SHFLX.daily.20060101-20801231.nc',
 'Q.monthly.200601-208012.nc',
 'TMQ.monthly.200601-208012.nc',
 'U.monthly.200601-208012.nc',
 'FSNTOA.daily.20060101-20801231.nc',
 'TREFHT.monthly.200601-208012.nc',
 'FLNS.monthly.192001-200512.nc',
 'FLNS.monthly.200601-208012.nc']
```

We can look at how big these files are…

[7]:
```
print("Original files")
for f in orig_files:
    print(
        f,
        " ",
        os.stat("/glade/p/cisl/asap/ldcpy_sample_data/lens/orig/" + f).st_size / 1e9,
```

(continues on next page)

(continued from previous page)

```
        "GB",
    )
```

```
Original files
CLOUD.monthly.200601-208012.nc   3.291160567 GB
LHFLX.daily.20060101-20801231.nc   4.804299285 GB
so4_a2_SRF.daily.20060101-20801231.nc   4.755841518 GB
PRECT.daily.20060101-20801231.nc   4.909326733 GB
TMQ.monthly.192001-200512.nc   0.160075734 GB
TAUX.daily.20060101-20801231.nc   4.856588097 GB
CCN3.monthly.200601-208012.nc   4.053932835 GB
PS.monthly.192001-200512.nc   0.12766304 GB
FLUT.daily.20060101-20801231.nc   4.091392046 GB
PRECT.daily.19200101-20051231.nc   5.629442994 GB
TS.daily.20060101-20801231.nc   3.435295036 GB
PS.monthly.200601-208012.nc   0.111186435 GB
TS.daily.19200101-20051231.nc   3.962086636 GB
Z500.daily.20060101-20801231.nc   3.244290942 GB
SHFLX.daily.20060101-20801231.nc   4.852980899 GB
Q.monthly.200601-208012.nc   3.809397495 GB
TMQ.monthly.200601-208012.nc   0.139301278 GB
U.monthly.200601-208012.nc   4.252600112 GB
FSNTOA.daily.20060101-20801231.nc   4.101201335 GB
TREFHT.monthly.200601-208012.nc   0.110713886 GB
FLNS.monthly.192001-200512.nc   0.170014618 GB
FLNS.monthly.200601-208012.nc   0.148417532 GB
```

Open datasets

First, let's look at the original and reconstructed files for the monthly surface Pressure (PS) data for 1920-2006. We begin by using ldcpy.open_dataset() to open the files of interest into our dataset collection. Usually we want chunks to be 100-150MB, but this is machine and app dependent.

[8]:
```
# load the first 86 years of montly surface pressure into a collection

col_PS = ldcpy.open_datasets(
    "cam-fv",
    ["PS"],
    [
        "/glade/p/cisl/asap/ldcpy_sample_data/lens/orig/PS.monthly.192001-200512.nc",
        "/glade/p/cisl/asap/ldcpy_sample_data/lens/lossy/c.PS.monthly.192001-200512.nc",
    ],
    ["orig", "lossy"],
    chunks={"time": 500},
)
col_PS
```

```
dataset size in GB 0.46
```

[8]:
```
<xarray.Dataset>
Dimensions:     (collection: 2, time: 1032, lat: 192, lon: 288)
```

(continues on next page)

(continued from previous page)

```
Coordinates:
  * lat         (lat) float64 -90.0 -89.06 -88.12 -87.17 ... 88.12 89.06 90.0
  * lon         (lon) float64 0.0 1.25 2.5 3.75 5.0 ... 355.0 356.2 357.5 358.8
  * time        (time) object 1920-02-01 00:00:00 ... 2006-01-01 00:00:00
    cell_area   (lat, collection, lon) float64 dask.array<chunksize=(192, 1, 288),␣
↪meta=np.ndarray>
  * collection  (collection) <U5 'orig' 'lossy'
Data variables:
    PS          (collection, time, lat, lon) float32 dask.array<chunksize=(1, 500, 192,␣
↪288), meta=np.ndarray>
Attributes: (12/15)
    Conventions:      CF-1.0
    source:           CAM
    case:             b.e11.B20TRC5CNBDRD.f09_g16.031
    title:            UNSET
    logname:          mickelso
    host:             ys0219
    ...               ...
    topography_file:  /glade/p/cesmdata/cseg/inputdata/atm/cam/topo/USGS-gtop...
    history:          Tue Nov  3 13:51:10 2020: ncks -L 5 PS.monthly.192001-2...
    NCO:              netCDF Operators version 4.7.9 (Homepage = http://nco.s...
    cell_measures:    area: cell_area
    data_type:        cam-fv
    file_size:        {'orig': 127663040, 'lossy': 39865015}
```

Data comparison

Now we use the ldcpy package features to compare the data.

Surface Pressure

Let's look at the comparison statistics at the first timeslice for PS.

[9]:
```
ps0 = col_PS.isel(time=0)
ldcpy.compare_stats(ps0, "PS", ["orig", "lossy"])
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

	orig	lossy
mean	98509	98493
variance	8.4256e+07	8.425e+07
standard deviation	9179.1	9178.8
min value	51967	51952
min (abs) nonzero value	51967	51952
max value	1.0299e+05	1.0298e+05
probability positive	1	1
number of zeros	0	0
spatial autocorr - latitude	0.98434	0.98434
spatial autocorr - longitude	0.99136	0.99136
entropy estimate	0.40644	0.11999

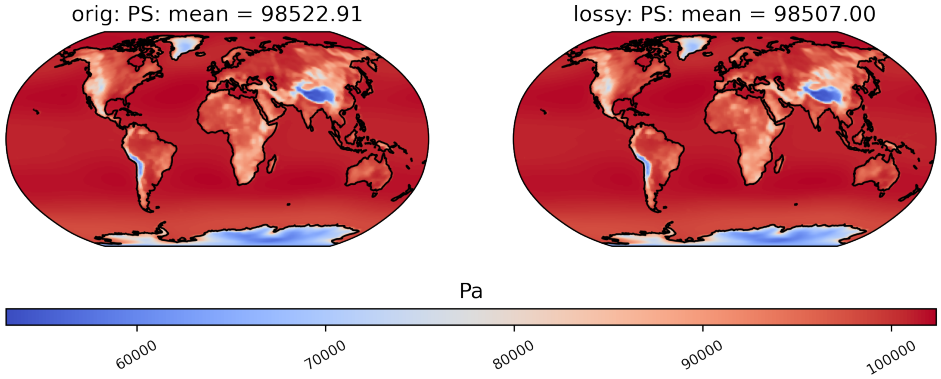
```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

	lossy
max abs diff	31.992
min abs diff	0
mean abs diff	15.906
mean squared diff	253.01
root mean squared diff	18.388
normalized root mean squared diff	0.0003587
normalized max pointwise error	0.00062698
pearson correlation coefficient	1
ks p-value	1.6583e-05
spatial relative error(% > 0.0001)	69.085
max spatial relative error	0.00048247
DSSIM	0.91512
file size ratio	3.2

Now we make a plot to compare the mean PS values across time in the orig and lossy datasets.

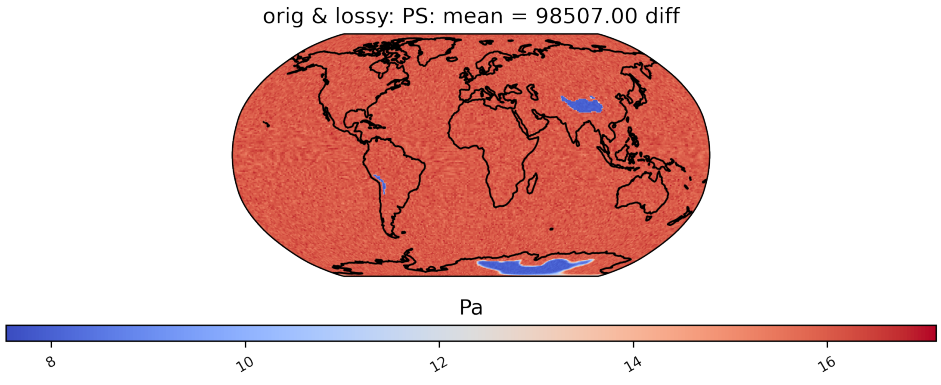
[10]:
```
# comparison between mean PS values (over the 86 years) in col_PS orig and lossy datasets
ldcpy.plot(col_PS, "PS", sets=["orig", "lossy"], calc="mean")
```



nbsphinx-code-borderwhite

Now we instead show the difference plot for the above plots.

[11]:
```
# diff between mean PS values in col_PS orig and lossy datasets
ldcpy.plot(col_PS, "PS", sets=["orig", "lossy"], calc="mean", calc_type="diff")
```

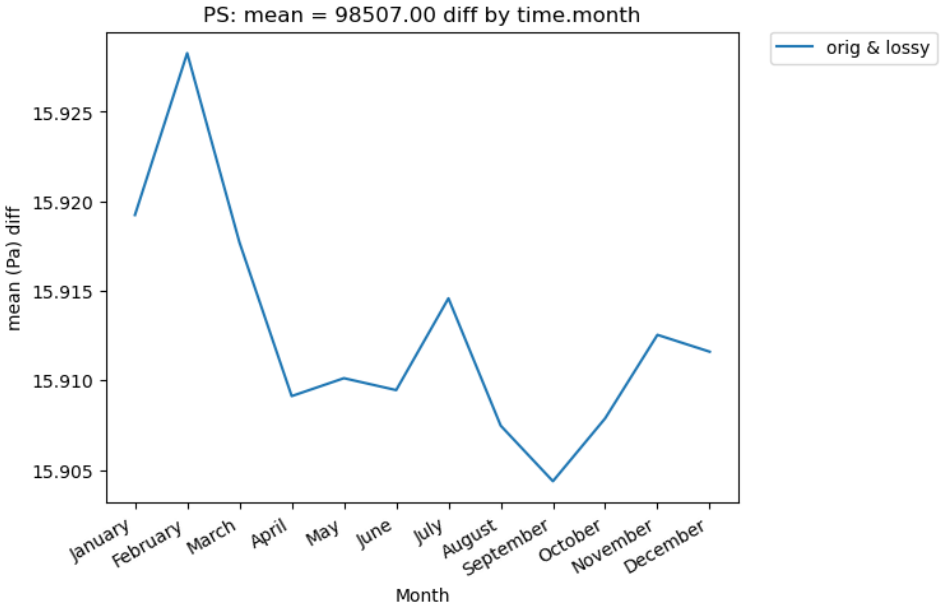


nbsphinx-code-borderwhite

We can also look at mean differences over time. Here we are looking at the spatial averages and then grouping by day

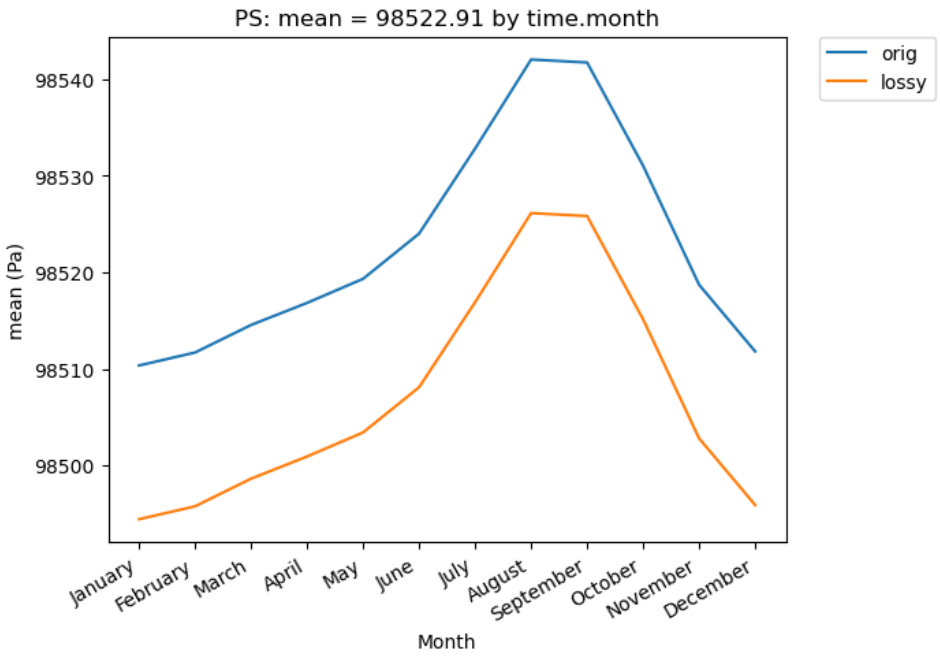
of the year. If doing a timeseries plot for this much data, using "group_by" is often a good idea.

[12]:
```
# Time-series plot of mean PS differences between col_PS orig and col_PS lossy datasets␣
↪grouped by month of year
ldcpy.plot(
    col_PS,
    "PS",
    sets=["orig", "lossy"],
    calc="mean",
    plot_type="time_series",
    group_by="time.month",
    calc_type="diff",
)
```



nbsphinx-code-borderwhite

[13]:
```
# Time-series plot of PS mean (grouped by month) in the original and lossy datasets
ldcpy.plot(
    col_PS,
    "PS",
    sets=["orig", "lossy"],
    calc="mean",
    plot_type="time_series",
    group_by="time.month",
)
```



nbsphinx-code-borderwhite

[14]:
```
del col_PS
```

Surface Temperature

Now let's open the daily surface temperature (TS) data for 1920-2005 into a collection. These are larger files than the monthly PS data.

[15]:
```
# load the first 86 years of daily surface temperature (TS) data
col_TS = ldcpy.open_datasets(
    "cam-fv",
    ["TS"],
    [
        "/glade/p/cisl/asap/ldcpy_sample_data/lens/orig/TS.daily.19200101-20051231.nc",
        "/glade/p/cisl/asap/ldcpy_sample_data/lens/lossy/c.TS.daily.19200101-20051231.nc
↪",
    ],
    ["orig", "lossy"],
    chunks={"time": 500},
)
col_TS
```

```
dataset size in GB 13.89
```

[15]:
```
<xarray.Dataset>
Dimensions:     (collection: 2, time: 31390, lat: 192, lon: 288)
Coordinates:
  * lat         (lat) float64 -90.0 -89.06 -88.12 -87.17 ... 88.12 89.06 90.0
  * lon         (lon) float64 0.0 1.25 2.5 3.75 5.0 ... 355.0 356.2 357.5 358.8
  * time        (time) object 1920-01-01 00:00:00 ... 2005-12-31 00:00:00
```

(continues on next page)

(continued from previous page)

```
    cell_area   (lat, collection, lon) float64 dask.array<chunksize=(192, 1, 288),␣
↪meta=np.ndarray>
  * collection  (collection) <U5 'orig' 'lossy'
Data variables:
    TS          (collection, time, lat, lon) float32 dask.array<chunksize=(1, 500, 192,␣
↪288), meta=np.ndarray>
Attributes: (12/15)
    Conventions:      CF-1.0
    source:           CAM
    case:             b.e11.B20TRC5CNBDRD.f09_g16.031
    title:            UNSET
    logname:          mickelso
    host:             ys0219
    ...               ...
    topography_file:  /glade/p/cesmdata/cseg/inputdata/atm/cam/topo/USGS-gtop...
    history:          Tue Nov  3 13:56:03 2020: ncks -L 5 TS.daily.19200101-2...
    NCO:              netCDF Operators version 4.7.9 (Homepage = http://nco.s...
    cell_measures:    area: cell_area
    data_type:        cam-fv
    file_size:        {'orig': 3962086636, 'lossy': 1330827000}
```

Look at the first time slice (time = 0) statistics:

[16]:
```
ldcpy.compare_stats(col_TS.isel(time=0), "TS", ["orig", "lossy"])
```

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

	orig	lossy
mean	284.49	284.43
variance	533.99	533.44
standard deviation	23.108	23.096
min value	216.73	216.69
min (abs) nonzero value	216.73	216.69
max value	315.58	315.5
probability positive	1	1
number of zeros	0	0
spatial autocorr - latitude	0.99392	0.99392
spatial autocorr - longitude	0.9968	0.9968
entropy estimate	0.41487	0.13675

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

	lossy
max abs diff	0.12497
min abs diff	0
mean abs diff	0.059427
mean squared diff	0.0035316
root mean squared diff	0.069462
normalized root mean squared diff	0.0006603
normalized max pointwise error	0.0012642
pearson correlation coefficient	1

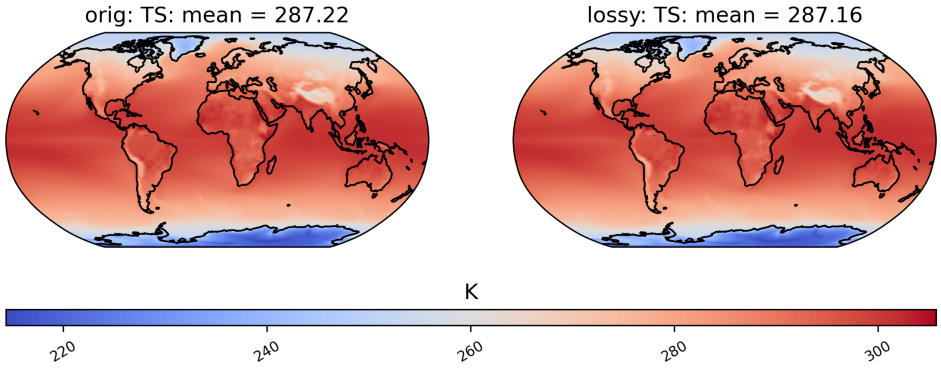
(continues on next page)

(continued from previous page)

```
ks p-value                              0.36817
spatial relative error(% > 0.0001)       73.293
max spatial relative error           0.00048733
DSSIM                                   0.97883
file size ratio                            2.98
```

Now we compare mean TS over time in a plot:

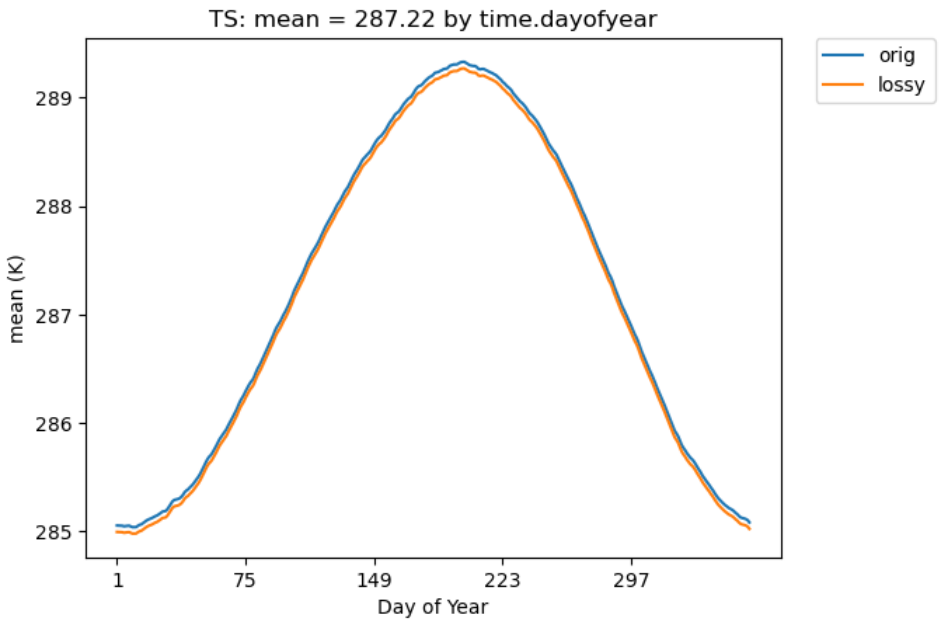
[17]:
```
# comparison between mean TS values in col_TS orig and lossy datasets
ldcpy.plot(col_TS, "TS", sets=["orig", "lossy"], calc="mean")
```



nbsphinx-code-borderwhite

Below we do a time series plot and group by day of the year. (Note that the group_by functionality is not fast.)

[18]:
```
# Time-series plot of TS means (grouped by days) in the original and lossy datasets
ldcpy.plot(
    col_TS,
    "TS",
    sets=["orig", "lossy"],
    calc="mean",
    plot_type="time_series",
    group_by="time.dayofyear",
)
```



nbsphinx-code-borderwhite

Let's delete the PS and TS data to free up memory.

[19]:
```
del col_TS
```

Precipitation Rate

Now let's open the daily precipitation rate (PRECT) data for 2006-2080 into a collection.

[20]:
```
# load the last 75 years of PRECT data
col_PRECT = ldcpy.open_datasets(
    "cam-fv",
    ["PRECT"],
    [
        "/glade/p/cisl/asap/ldcpy_sample_data/lens/orig/PRECT.daily.20060101-20801231.nc
↪",
        "/glade/p/cisl/asap/ldcpy_sample_data/lens/lossy/c.PRECT.daily.20060101-20801231.
↪nc",
    ],
    ["orig", "lossy"],
    chunks={"time": 500},
)
col_PRECT
```

```
dataset size in GB 12.11
```

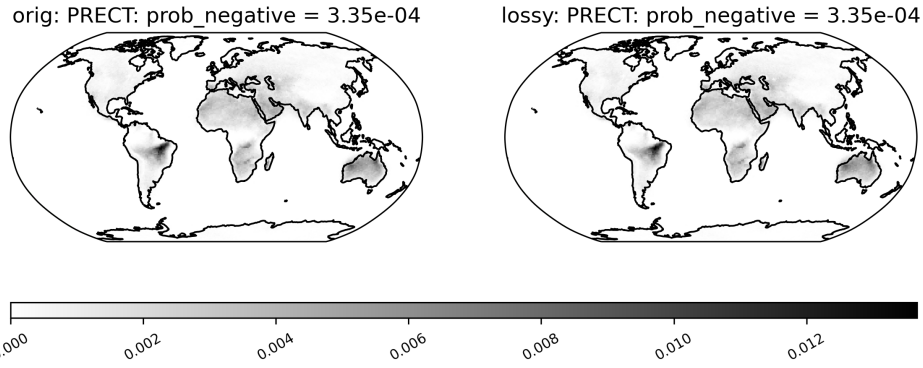
[20]:
```
<xarray.Dataset>
Dimensions:     (collection: 2, time: 27375, lat: 192, lon: 288)
Coordinates:
  * lat         (lat) float64 -90.0 -89.06 -88.12 -87.17 ... 88.12 89.06 90.0
  * lon         (lon) float64 0.0 1.25 2.5 3.75 5.0 ... 355.0 356.2 357.5 358.8
  * time        (time) object 2006-01-01 00:00:00 ... 2080-12-31 00:00:00
```

(continues on next page)

(continued from previous page)

```
    cell_area   (lat, collection, lon) float64 dask.array<chunksize=(192, 1, 288),␣
↪meta=np.ndarray>
  * collection  (collection) <U5 'orig' 'lossy'
Data variables:
    PRECT       (collection, time, lat, lon) float32 dask.array<chunksize=(1, 500, 192,␣
↪288), meta=np.ndarray>
Attributes: (12/15)
    Conventions:      CF-1.0
    source:           CAM
    case:             b.e11.BRCP85C5CNBDRD.f09_g16.031
    title:            UNSET
    logname:          mickelso
    host:             ys1023
    ...               ...
    topography_file:  /glade/p/cesmdata/cseg/inputdata/atm/cam/topo/USGS-gtop...
    history:          Tue Nov  3 14:13:51 2020: ncks -L 5 PRECT.daily.2006010...
    NCO:              netCDF Operators version 4.7.9 (Homepage = http://nco.s...
    cell_measures:    area: cell_area
    data_type:        cam-fv
    file_size:        {'orig': 4909326733, 'lossy': 3446890300}
```

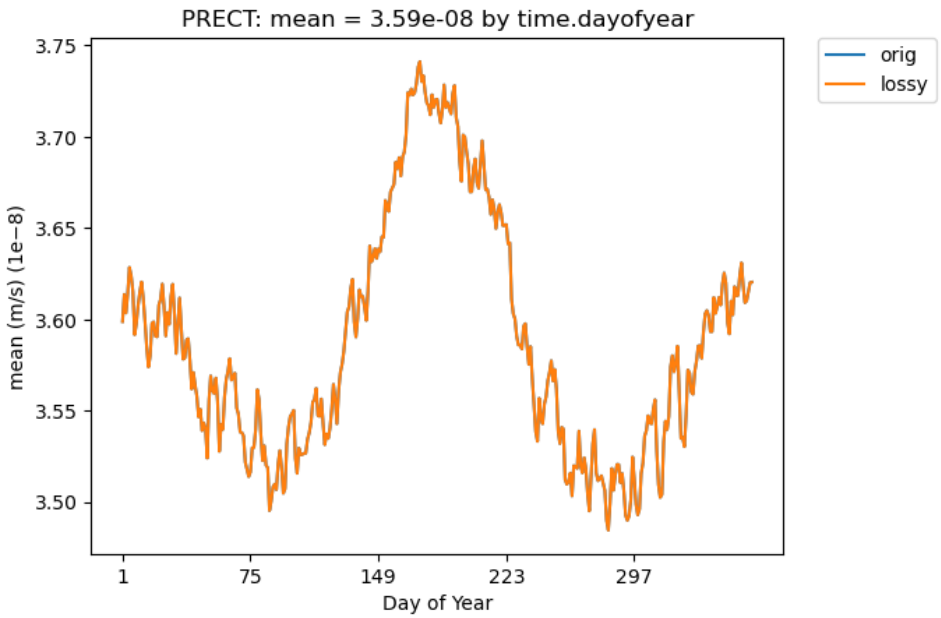
[21]:
```
# compare probability of negative rainfall (and get ssim)
ldcpy.plot(col_PRECT, "PRECT", sets=["orig", "lossy"], calc="prob_negative", color=
↪"binary")
```



nbsphinx-code-borderwhite

Mean PRECT over time…

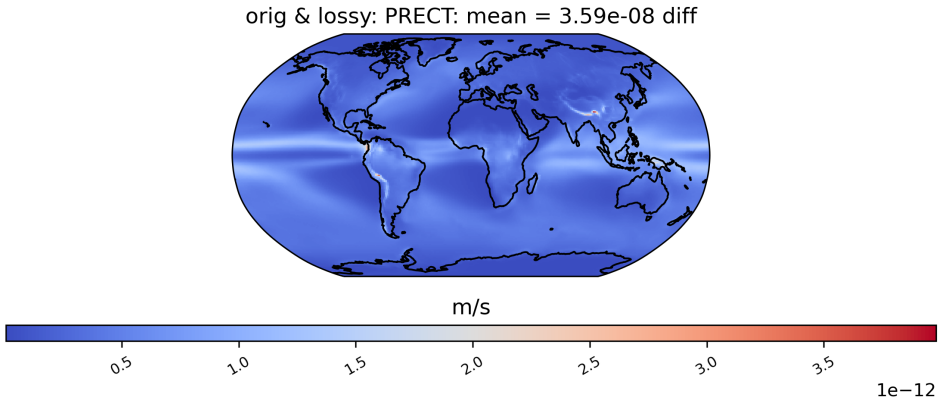
[22]:
```
# Time-series plot of PRECT mean in 'orig' dataset
ldcpy.plot(
    col_PRECT,
    "PRECT",
    sets=["orig", "lossy"],
    calc="mean",
    plot_type="time_series",
    group_by="time.dayofyear",
)
```



nbsphinx-code-borderwhite

Now look at mean over time spatial plot:

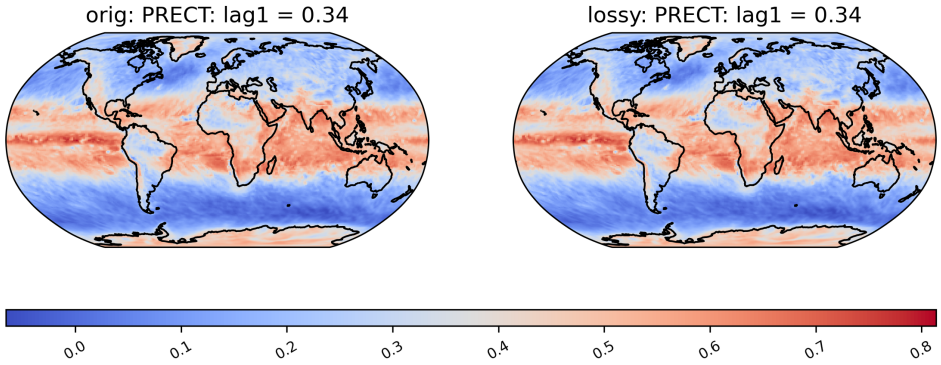
[23]:
```
# diff between mean PRECT values across the entire timeseries
ldcpy.plot(
    col_PRECT,
    "PRECT",
    sets=["orig", "lossy"],
    calc="mean",
    calc_type="diff",
)
```



nbsphinx-code-borderwhite

Calculating the correlation of the lag-1 values … for the first 10 years (This operation can be time-consuming).

[24]:
```
# plot of lag-1 correlation of PRECT values
ldcpy.plot(col_PRECT, "PRECT", sets=["orig", "lossy"], calc="lag1", start=0, end=3650)
```



nbsphinx-code-borderwhite

CAGEO Plots

Plots from different data sets used in CAGEO paper (Poppick et al., "A Statistical Analysis of Lossily Compressed Climate Model Data",doi:10.1016/j.cageo.2020.104599) comparing the sz and zfp compressors with a number of metrics (specified with "calc" and "calc_type" in the plot routine).

[25]:
```
col_TS = ldcpy.open_datasets(
    "cam-fv",
    ["TS"],
    [
        "/glade/p/cisl/asap/ldcpy_sample_data/cageo/orig/b.e11.B20TRC5CNBDRD.f09_g16.030.
↪cam.h1.TS.19200101-20051231.nc",
        "/glade/p/cisl/asap/ldcpy_sample_data/cageo/lossy/sz1.0.TS.nc",
        "/glade/p/cisl/asap/ldcpy_sample_data/cageo/lossy/zfp1.0.TS.nc",
    ],
    ["orig", "sz1.0", "zfp1.0"],
    chunks={"time": 500},
)
col_PRECT = ldcpy.open_datasets(
    "cam-fv",
    ["PRECT"],
    [
        "/glade/p/cisl/asap/ldcpy_sample_data/cageo/orig/b.e11.B20TRC5CNBDRD.f09_g16.030.
↪cam.h1.PRECT.19200101-20051231.nc",
        "/glade/p/cisl/asap/ldcpy_sample_data/cageo/lossy/sz1e-8.PRECT.nc",
        "/glade/p/cisl/asap/ldcpy_sample_data/cageo/lossy/zfp1e-8.PRECT.nc",
    ],
    ["orig", "sz1e-8", "zfp1e-8"],
    chunks={"time": 500},
)
```

```
dataset size in GB 20.83

dataset size in GB 20.83
```

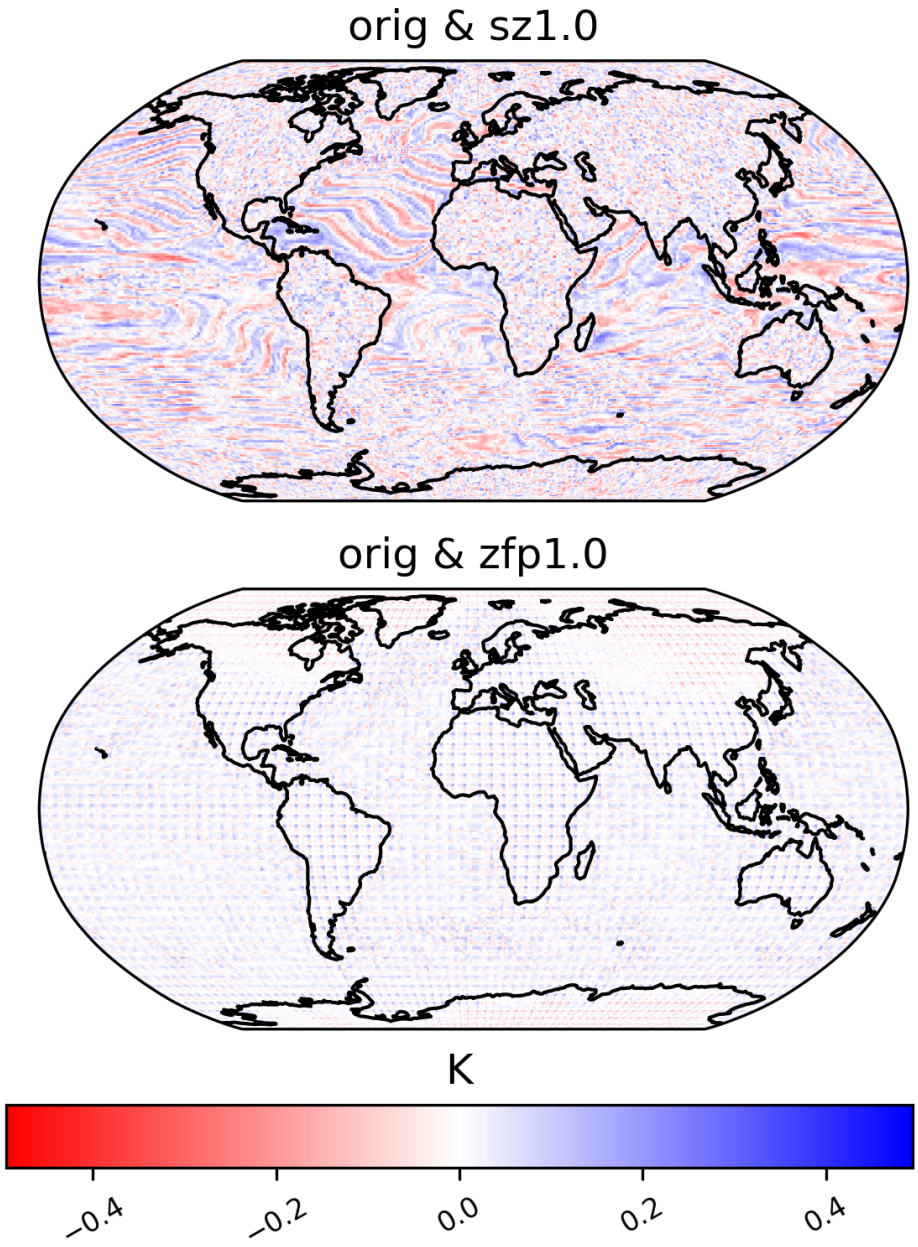
difference in mean

[26]:
```
ldcpy.plot(
    col_TS,
```

(continues on next page)

(continued from previous page)

```
    "TS",
    sets=["orig", "sz1.0", "zfp1.0"],
    calc="mean",
    calc_type="diff",
    color="bwr_r",
    start=0,
    end=50,
    tex_format=False,
    vert_plot=True,
    short_title=True,
    axes_symmetric=True,
)
```



nbsphinx-code-borderwhite

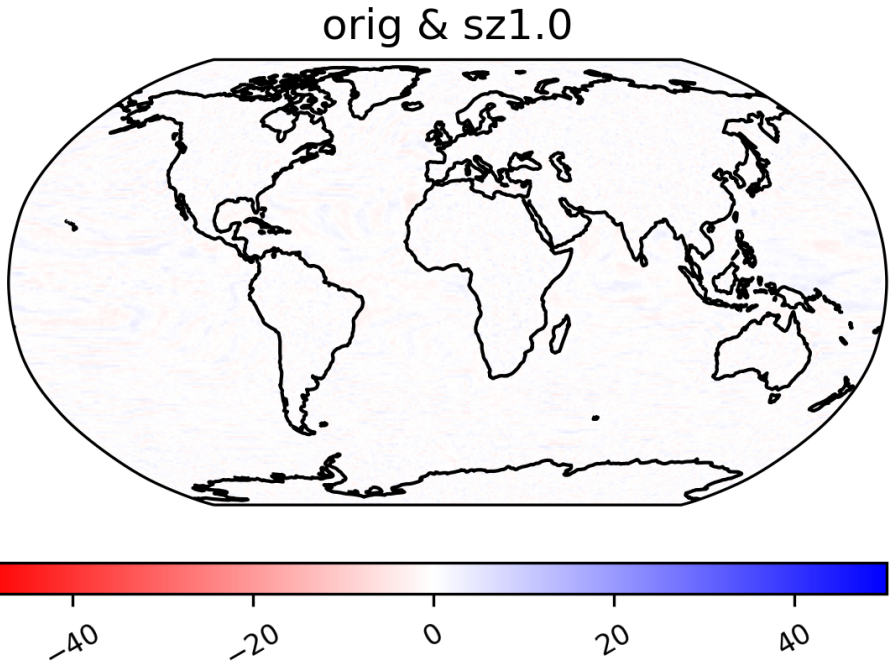
Zscore

[27]:
```
ldcpy.plot(
    col_TS,
    "TS",
    sets=["orig", "sz1.0"],
    calc="zscore",
    calc_type="calc_of_diff",
    color="bwr_r",
    start=0,
    end=1000,
    tex_format=False,
```

(continues on next page)

(continued from previous page)

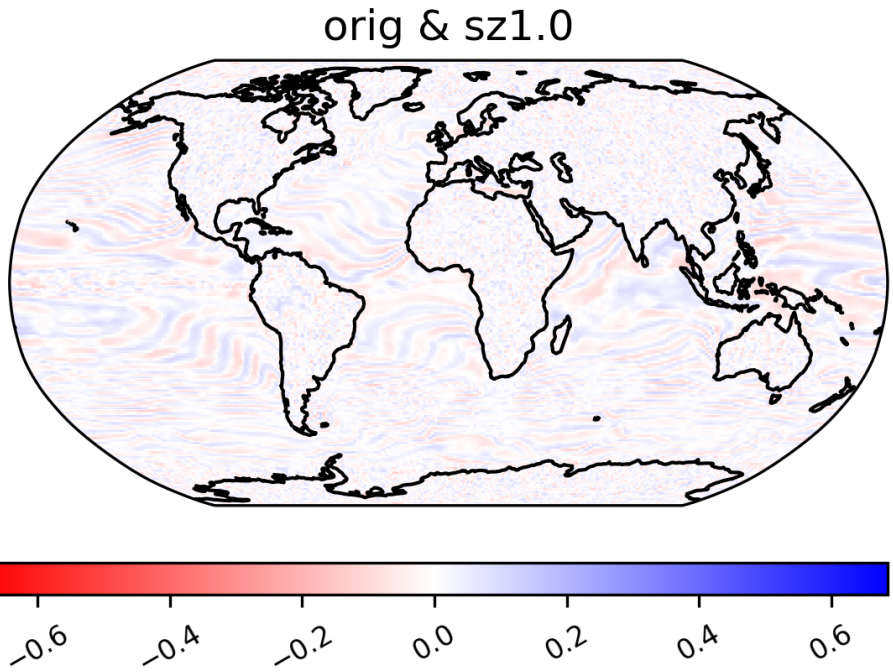
```
    vert_plot=True,
    short_title=True,
    axes_symmetric=True,
)
```



nbsphinx-code-borderwhite

Pooled variance ratio

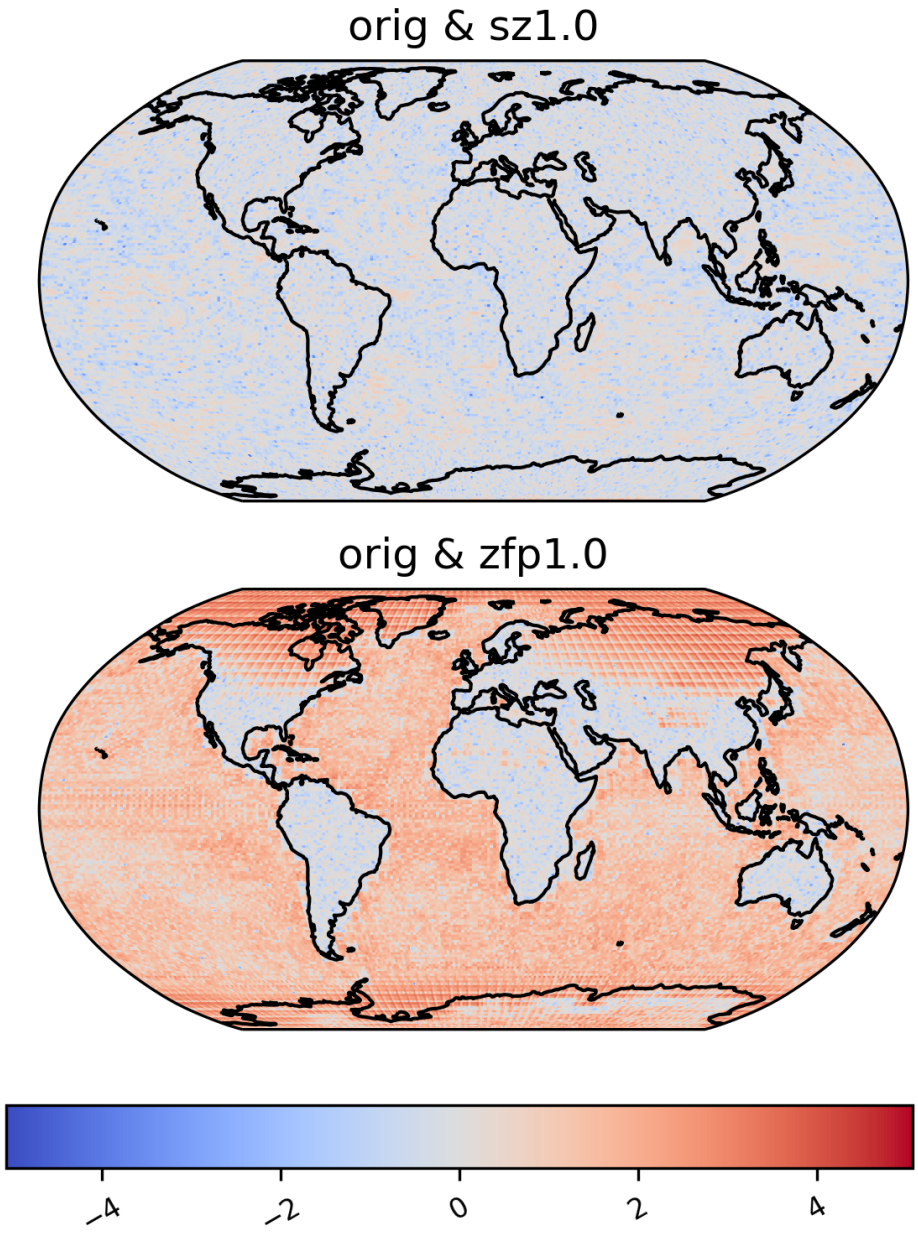
[28]:
```
ldcpy.plot(
    col_TS,
    "TS",
    sets=["orig", "sz1.0"],
    calc="pooled_var_ratio",
    color="bwr_r",
    calc_type="calc_of_diff",
    transform="log",
    start=0,
    end=100,
    tex_format=False,
    vert_plot=True,
    short_title=True,
    axes_symmetric=True,
)
```



nbsphinx-code-borderwhite

annual harmonic relative ratio

[29]:
```
col_TS["TS"] = col_TS.TS.chunk({"time": -1, "lat": 10, "lon": 10})
ldcpy.plot(
    col_TS,
    "TS",
    sets=["orig", "sz1.0", "zfp1.0"],
    calc="ann_harmonic_ratio",
    transform="log",
    calc_type="calc_of_diff",
    tex_format=False,
    axes_symmetric=True,
    vert_plot=True,
    short_title=True,
)
col_TS["TS"] = col_TS.TS.chunk({"time": 500, "lat": 192, "lon": 288})
```



nbsphinx-code-borderwhite

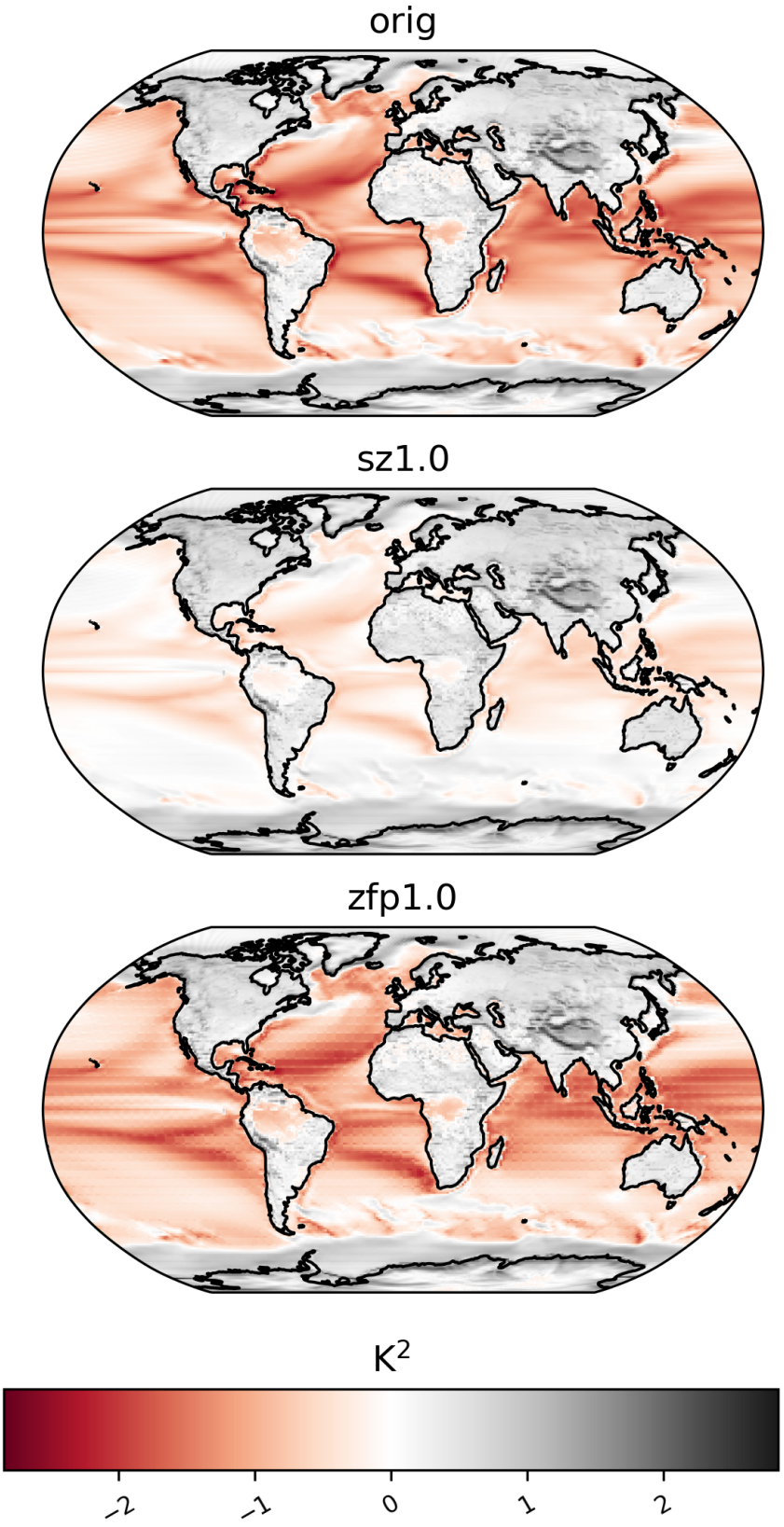
NS contrast variance

[30]:
```
ldcpy.plot(
    col_TS,
    "TS",
    sets=["orig", "sz1.0", "zfp1.0"],
    calc="ns_con_var",
    color="RdGy",
    calc_type="raw",
    transform="log",
    axes_symmetric=True,
    tex_format=False,
```

(continues on next page)

(continued from previous page)

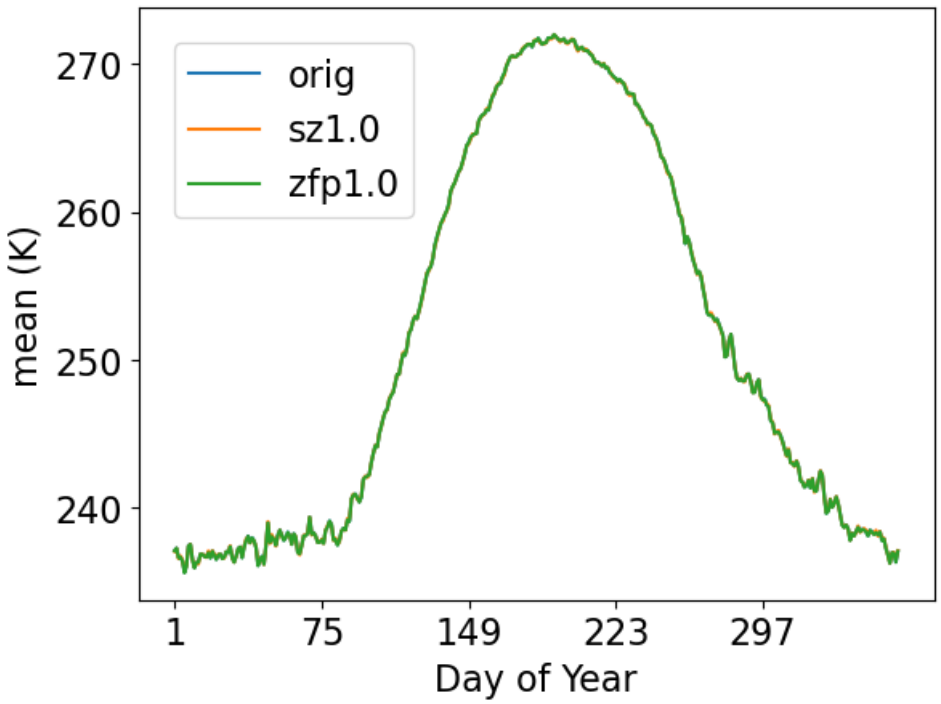
```
    vert_plot=True,
    short_title=True,
)
```



nbsphinx-code-borderwhite

mean by day of year

[31]:
```
ldcpy.plot(
    col_TS,
    "TS",
    sets=["orig", "sz1.0", "zfp1.0"],
    calc="mean",
    plot_type="time_series",
    group_by="time.dayofyear",
    legend_loc="upper left",
    lat=90,
    lon=0,
    vert_plot=True,
    tex_format=False,
    short_title=True,
)
```



nbsphinx-code-borderwhite

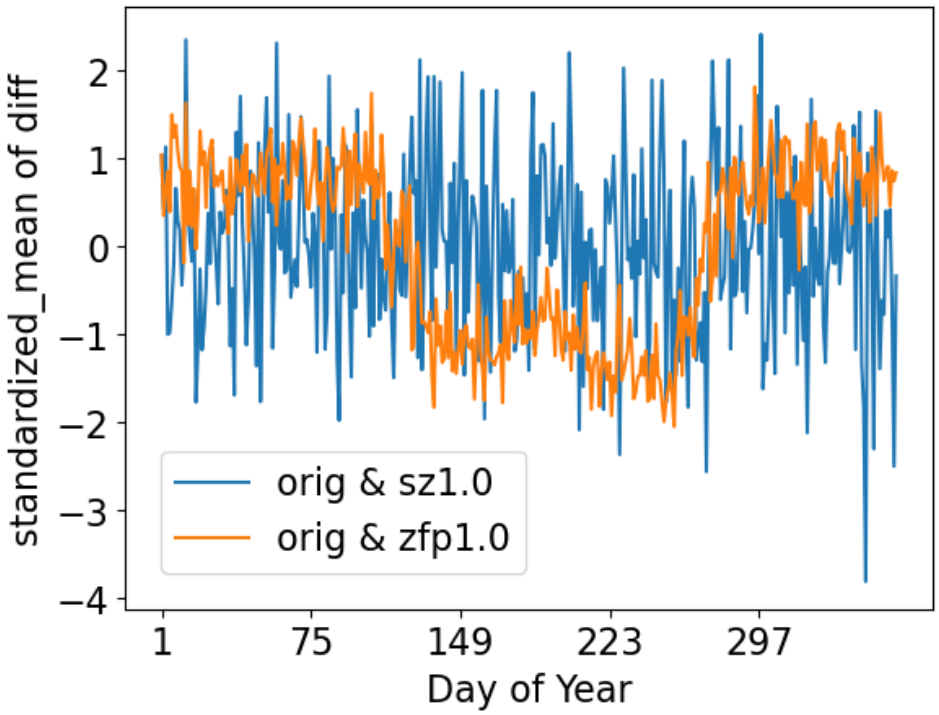
standardized mean errors by day of year

[32]:
```
ldcpy.plot(
    col_TS,
    "TS",
    sets=["orig", "sz1.0", "zfp1.0"],
    calc="standardized_mean",
    legend_loc="lower left",
    calc_type="calc_of_diff",
    plot_type="time_series",
    group_by="time.dayofyear",
    tex_format=False,
    lat=90,
    lon=0,
```

(continues on next page)

(continued from previous page)

```
    vert_plot=True,
    short_title=True,
)
```



nbsphinx-code-borderwhite

[33]:
```
del col_TS
```

Do any other comparisons you wish … and then clean up!

[34]:
```
client.close()
```

[ ]:
```
```

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

R

S

T

V

W

Z